
ACTA CYBERNETICA

Editor-in-Chief: János Csirik (Hungary)

Managing Editor: Csanád Imreh (Hungary)

Assistant to the Managing Editor: Attila Tanács (Hungary)

Associate Editors:

Luca Aceto (Iceland)

Mátyás Arató (Hungary)

Hans L. Bodlaender (The Netherlands)

Horst Bunke (Switzerland)

Tibor Csendes (Hungary)

János Demetrovics (Hungary)

Bálint Dömölki (Hungary)

Zoltán Ésik (Hungary)

Zoltán Fülöp (Hungary)

Ferenc Gécseg (Hungary)

Jozef Gruska (Slovakia)

Tibor Gyimóthy (Hungary)

Helmut Jürgensen (Canada)

Zoltan Kato (Hungary)

Alice Kelemenová (Czech Republic)

László Lovász (Hungary)

Gheorghe Păun (Romania)

András Prékopa (Hungary)

Arto Salomaa (Finland)

László Varga (Hungary)

Heiko Vogler (Germany)

Gerhard J. Woeginger (The Netherlands)

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. An electronic version of the published paper is provided for the authors in PDF format.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in L^AT_EX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu

Web access. The above informations along with the contents of past issues are available at the Acta Cybernetica homepage <http://www.inf.u-szeged.hu/actacybernetica/>.

EDITORIAL BOARD

Editor-in-Chief: **János Csirik**

Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
csirik@inf.u-szeged.hu

Managing Editor: **Csanád Imreh**

Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
cimreh@inf.u-szeged.hu

Assistant to the Managing Editor:

Attila Tanács

Department of Image Processing
and Computer Graphics
University of Szeged, Szeged, Hungary
tanacs@inf.u-szeged.hu

Associate Editors:

Luca Aceto

School of Computer Science
Reykjavík University
Reykjavík, Iceland
luca@ru.is

Mátyás Arató

Faculty of Informatics
University of Debrecen
Debrecen, Hungary
arato@inf.unideb.hu

Hans L. Bodlaender

Institute of Information and
Computing Sciences
Utrecht University
Utrecht, The Netherlands
hansb@cs.uu.nl

Horst Bunke

Institute of Computer Science and
Applied Mathematics
University of Bern
Bern, Switzerland
bunke@iam.unibe.ch

Tibor Csendes

Department of Applied Informatics
University of Szeged
Szeged, Hungary
csendes@inf.u-szeged.hu

János Demetrovics

MTA SZTAKI
Budapest, Hungary
demetrovics@sztaki.hu

Bálint Dömölki

John von Neumann Computer Society
Budapest, Hungary

Zoltán Ésik

Department of Foundations of
Computer Science
University of Szeged
Szeged, Hungary
ze@inf.u-szeged.hu

Zoltán Fülöp

Department of Foundations of
Computer Science
University of Szeged
Szeged, Hungary
fulop@inf.u-szeged.hu

Ferenc Gécseg
Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
gecseg@inf.u-szeged.hu

Jozef Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Bratislava, Slovakia
gruska@savba.sk

Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

Helmut Jürgensen
Department of Computer Science
Middlesex College
The University of Western Ontario
London, Canada
helmut@csd.uwo.ca

Zoltan Kato
Department of Image Processing
and Computer Graphics
Szeged, Hungary
kato@inf.u-szeged.hu

Alice Kelemenová
Institute of Computer Science
Silesian University at Opava
Opava, Czech Republic
Alica.Kelemenova@fpf.slu.cz

László Lovász
Department of Computer Science
Eötvös Loránd University
Budapest, Hungary
lovasz@cs.elte.hu

Gheorghe Păun
Institute of Mathematics of the
Romanian Academy
Bucharest, Romania
George.Paun@imar.ro

András Prékopa
Department of Operations Research
Eötvös Loránd University
Budapest, Hungary
prekopa@cs.elte.hu

Arto Salomaa
Department of Mathematics
University of Turku
Turku, Finland
asalomaa@utu.fi

László Varga
Department of Software Technology
and Methodology
Eötvös Loránd University
Budapest, Hungary
varga@ludens.elte.hu

Heiko Vogler
Department of Computer Science
Dresden University of Technology
Dresden, Germany
Heiko.Vogler@tu-dresden.de

Gerhard J. Woeginger
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
gwoegi@win.tue.nl

SYMPOSIUM ON PROGRAMMING LANGUAGES AND SOFTWARE TOOLS (SPLST'13)

Selected Papers from the Symposium

Guest Editor:

Ákos Kiss

Department of Software Engineering
University of Szeged
Szeged, Hungary
akiss@inf.u-szeged.hu

Preface

This special issue contains papers on topics of the 13th Symposium on Programming Languages and Software Tools (SPLST'13). The series started in 1989 in Szeged, Hungary, and since then, by tradition, it has been organized every second year in Hungary, Finland, and Estonia, with participants coming from all over Europe. The thirteenth edition of the symposium was back again in Szeged on August 26–27, 2013, organized by the Department of Software Engineering of the University of Szeged. The purpose of the SPLST has always been to provide a forum for software scientists to present and discuss recent researches and developments related to programming languages, software tools, and methods for software development. At SPLST'13, there were 20 accepted talks in sections on program analysis, formal verification, software evolution and maintenance, and web technologies. In addition, an invited talk was presented jointly by Hassan Charaf and László Lengyel (Budapest University of Technology and Economics). After the symposium, the authors of selected talks were invited to revise and extend their papers for publication in *Acta Cybernetica*. Following a review process, 9 were accepted for publication, which are presented in this special issue.

Ákos Kiss
Guest Editor

Lively3D: Building a 3D Desktop Environment as a Single Page Application

Jari-Pekka Voutilainen*, Anna-Liisa Mattila*, and Tommi Mikkonen*

Abstract

The Web has rapidly evolved from a simple document browsing and distribution environment into a rich software platform, where desktop-style applications are treated as first class citizens. Despite the associated technical complexities and limitations, it is not unusual to find complex applications that build on the web as their only platform, with no traditional installable application for the desktop environment – such systems are simply accessed via a web page that is downloaded inside the browser and once loading is completed, the application will begin its execution immediately. With the recent standardization efforts, including HTML5 and WebGL in particular, compelling, visually rich applications are increasingly supported by the browsers. In this paper, we demonstrate the new facilities of the browser as a visualization tool, going beyond what is expected of traditional web applications. In particular, we demonstrate that with mashup technologies, which enable combining already existing content from various sites into an integrated experience, the new graphics facilities unleashes unforeseen potential for visualizations.

Keywords: web apps, visualization, window management, 3D UI

1 Introduction

Over the few recent years, the Web has evolved from a simple document browsing and distribution environment into a rich software platform, which is capable of hosting desktop-style applications. Moreover, these applications are increasingly often treated as first class citizens.

The document-centric origins of the Web are still visible in many areas. Consequently, it has been traditionally considered difficult to compose truly interactive web applications. A partial solution has been to use plug-in components or browser extensions, such as Adobe Flash or Microsoft Silverlight, but such binary or company specific technologies do not fit well to the ideals of the open web, advocating web applications that are built using technologies that are open, accessible and as

*Department of Pervasive Computing, Tampere University of Technology, E-mail: {jari-pekka.voutilainen, anna-liisa.mattila, tommi.mikkonen}@tut.fi

interoperable as possible to avoid vendor-specific lock-in. As a manifestation of this attitude, it is not unusual for complex applications to use the web as their only platform. In other words, despite the technical difficulties and limitations, there is no traditional installable application for the desktop – the system is simply accessed via a web page that is downloaded inside the browser, whose runtime resources are then used by the application. We believe that the transition of applications from the desktop computer to the web has only started, and the variety, number, and importance of web applications will be constantly rising during the next several years to come.

In comparison to desktop applications, the benefits of web applications are many. Web applications are easy to adopt, because they need neither installation nor updating – one simply enters the URL into the browser and the latest version is always run. Furthermore, web applications are easy and cheap to publish and maintain; there is no need for intermediates like shops or distributors. Furthermore, in comparison to conventional desktop applications, web applications have a whole new set of features available, like online collaboration, user created content, shared data, and distributed workspace. Finally, with the whole content of the web acting as the data repository, the new application development opportunities, unleashed by the newly introduced facilities of the web technologies that make the browser increasingly capable platform for running interactive applications, are increasing the potential of the web as an application platform.

In this paper, we demonstrate the new facilities of the web as an information visualization tool, going beyond what is expected of browser based applications. Moreover, we demonstrate that together with mashup technologies, which enable combining already existing content from various sites into an integrated, usually more compelling experience, the new graphics facilities results in unforeseen potential for visualization of context-specific data. Together with data science, the approach can be generalized to increasingly complex systems, which simplifies data consumption tremendously.

The rest of the paper is structured as follows. In Section 2, we discuss the evolution of the web and the main phases that can be identified in the process, and briefly address two important web standards - HTML5 and WebGL - and their role in the development of new types of web applications, building on already available resources. In Section 3, we introduce our technical contribution, Lively3D, which is a host environment that is capable of integrating multiple applications within single 3D-scene and visualize the environment in three different ways. In Section 4, we discuss development issues related to Lively3D's 3D user interface and introduce a redesigned version of Lively3D's UI. In Section 5 final conclusions are drawn.

2 Background

The World Wide Web has undergone a number of evolutionary phases [6]. In the first phase, web pages were truly pages, and navigation between pages was based simply on hyperlinks – a new web page was loaded from the web server each time

the user clicked on a link. These pages were truly page-structured documents that contained primarily text with some interspersed static images, without animation or any interactive content, which were only introduced in the second phase, as web pages became increasingly interactive, created by using animated graphics and plugin components. In this phase, the JavaScript scripting language enabled building animated, interactive content with technologies primarily associated with the Web only. Moreover, as a part of the transition to this phase, the Web started moving in directions that were unforeseen by its designers. Web sites started behaving more like multimedia presentations rather than page-structured documents, content mashups and web site cross-linking became increasingly popular.

Today, the browser is increasingly used as a platform for real applications, with services such as Google Docs with its desktop-like interactions paving the way towards more complex systems. We expect that as more and more data becomes available online, the capabilities of the browser will be increasingly often harnessed to filter and further process the data into a form that can be more easily consumed. In this context, two recent initiatives form an important perspective. These are the open web, perhaps best manifested in Mozilla Manifesto¹, which centers around the idea that the web that is a global public resource that must remain open, accessible, interoperable and secure, and open data, which according to Wikipedia², builds on the idea that certain data should be freely available to everyone to use and republish as they wish, without restrictions from copyrights, patents, or other mechanisms of control.

To support the above initiatives, the need to use plugins is being seriously challenged by two recently introduced technologies, HTML5 and WebGL, as already pointed out in [5]. These new technologies provide support for creating desktop-like applications that run inside the browser (HTML5) and enable direct access to graphics facilities from web pages (WebGL). This, together with already well-known techniques for mashupping, are paving the way towards the next generation of web applications, with increasing capabilities for modeling and visualizing data and conceptual information.

The forthcoming HTML5 standard³ complements the capabilities of the existing HTML standard with numerous new features. Although HTML5 is a general-purpose web standard, many of the new features are aimed squarely at making the Web a better place for desktop-style web applications. There are numerous additions when compared to the earlier versions of the HTML specification. To begin with, the new standard will extend the set of available markup tags with important new elements. These new elements make it possible, e.g., to embed audio and video directly into web pages. This will eliminate the need to use plugin components such as Flash for such types of media. The HTML5 standard will also introduce various new interfaces and APIs that will be available for JavaScript applications.

¹<http://www.mozilla.org/about/manifesto.html>

²http://en.wikipedia.org/wiki/Open_data

³<http://www.w3.org/TR/html5/>

WebGL⁴ is a cross-platform web standard for hardware accelerated 3D graphics API developed by Mozilla, Khronos Group, and a consortium of additional companies including Apple, Google and Opera. The main feature that WebGL brings to the Web is the ability to display 3D graphics natively in the web browser without any plug-in components. WebGL is based on OpenGL ES 2.0⁵, and it uses the OpenGL shading language GLSL. WebGL runs in the HTML5's canvas element, and WebGL data is generally accessible through the web browser's Document Object Model (DOM) interface. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to JavaScript programmers.

As a technical detail, it is important to notice that the WebGL API is implemented at a lower level compared to the equivalent OpenGL APIs. This increases the software developers' burden as they have to implement some commonly used OpenGL functionality themselves. To make it easier and faster to use WebGL, several additional JavaScript frameworks and APIs have been introduced, including Three.js⁶, Copperlicht⁷, GLGE⁸, SceneJS⁹, and SpiderGL¹⁰. Such frameworks introduce their own JavaScript API through which the lower-level WebGL API is used. The goal of these libraries is to hide the majority of technical details and thus make it simpler to write applications using the framework APIs. Furthermore, these WebGL frameworks provide functions for performing basic 2D and 3D rendering operations such as drawing a rotating cube on the canvas. The more advanced libraries also have functions for performing animations, adding lighting and shadows, calculating the level of detail, collision detection, object selection, and so forth.

3 Lively3D: Host environment for web apps

The goal of the Lively 3D proof-of-concept design was to create a 3D environment in which applications of different kind – including data processing, visualization, and interactive applications in particular – can be embedded as separate elements within a single environment running inside the browser. Furthermore, the design is based on using facilities that are commonly used in the web already, implying that to a large extent it is possible to immediately reuse already existing content in the system.

⁴<http://www.khronos.org/webgl/>

⁵<http://www.khronos.org/opengles>

⁶<http://threejs.org/>

⁷<http://www.ambiera.com/copperlicht/>

⁸<http://www.glge.org/>

⁹<http://scenejs.org/>

¹⁰<http://spidergl.org/>

3.1 Overview

Web app, by simple definition¹¹, is an application utilizing web and [web] browser technologies to accomplish one or more tasks over a network, typically through [web] browser. Canvas application is a subset of web app, which uses a single canvas html element¹² as its graphical interface.

Lively3D¹³ is a web application framework, where embedded canvas applications are displayed inside a three dimensional windowing environment. Individual applications embedded in the system can thus be composed using the Canvas API, offered by HTML5. In general, this enables the creation of graphically rich small apps that are capable of interacting with the user in a desktop like fashion.

The conceptual idea of Lively3D is based on previous project *The Lively Kernel*[5]. Lively Kernel was 2D window manager and IDE that was executed in the browser. Similar frameworks and tools have been developed by others like Ventus¹⁴ and SproutCore¹⁵.

The Lively3D framework itself is implemented with GLGE¹⁶, a WebGL library by Paul Brunt, which abstracts numerous implementation details of WebGL from the developer. Embedding the applications to the framework was designed in such a way that the developer of a canvas application needs to implement minimal interfaces towards the Lively3D system in order to integrate the application within the environment. Existing canvas applications are easily converted to Lively3D app by wrapping the existing code to the Lively3D interfaces.

In addition to the applications, the 3D environment that displays the applications can be redefined using Lively3D interfaces. The applications and different 3D environments are deployed in a shared Dropbox folder, so that multiple developers can collaborate in implementing applications and environments without constantly updating the files on the server hosting Lively3D.

Lively3D is implemented as Single-Page Application (SPA) where the whole application is loaded with a single page load. This provides the user interface and the basic mechanics of 3D environments. SPA design was selected, so that applications can interact with the windowing environment and the whole state of the environment is stored within the JavaScript namespace. The design of Lively3D was considerably affected by the browser security model, which limits the possibilities of resource usage. The security model denies access both to the local file system and external resources in different domain with its Same-origin policy¹⁷. The policy is upheld in Lively3D with server-side proxies, so that the browser sees all the content in same domain. The main components of the system are illustrated in Figure 1. All components are designed with easy-to-use interfaces and require minimal knowledge of inner working of the framework.

¹¹<http://web.appstorm.net/general/opinion/what-is-a-web-app-heres-our-definition/>

¹²<http://www.w3.org/wiki/HTML/Elements/canvas>

¹³<http://lively3d.cs.tut.fi/>

¹⁴<http://www.rlamana.es/ventus/>

¹⁵<http://sproutcore.com/>

¹⁶<http://www.glge.org/>

¹⁷http://www.w3.org/Security/wiki/Same-Origin_Policy

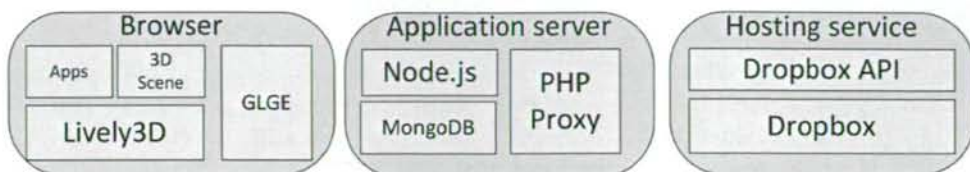


Figure 1: Structure of the Lively3D framework

Applications and 3D scenes are developed in JavaScript using Lively3D API, deployed to Dropbox using the official Dropbox client, and downloaded into Lively3D through PHP or Node.js proxies, depending on the situation. The Lively3D API provides resource loaders, which enable deployment of application and 3D-scene specific resources to the Dropbox so that complete applications and 3D scenes can be downloaded through the server hosting Lively3D, thus in essence circumventing browser security restrictions.

When a new 3D scene is designed and implemented, the developer has to define the essential functions that are called by the Lively3D environment, similarly to many other graphical user interface frameworks. These functions enable redefining how the system interacts with the user, including mouse interaction, the creation of 3D objects in the GLGE system that represents the application, and automatic updates of the scene between frames. Additionally, the initial state of the scene is defined in GLGE's XML format, which can be generated with 3D modeling software, like Blender (<http://www.blender.org/>) for example.

3.2 Lively3D apps

A Lively3D app consists of canvas application and its data structures in Lively3D host environment. Usable existing web apps are limited to canvas applications, because Lively3D is implemented in WebGL and the WebGL specification permits the use of canvas, image and video html-elements as the only source for textures within the 3D-environment. Most of the data structures are provided by Lively3D, but some conventions must be followed when converting existing canvas application to Lively3D app.

Since web apps are usually developed with expectancy that the app will be the only app in web page, the app structure can be pretty much anything the developer desires. But since Lively3D is implemented in Single Page Application paradigm, Lively3D apps are separated from each other with simulated namespaces as much as the browser model permits.

To achieve the above goal, each canvas application must have clearly separated initialization code. Additionally all the browser elements the app uses, must be created dynamically with a single canvas-element functioning as the only graphical element of the application. To mitigate these restrictions Lively3D offers API for canvas applications, which is presented in figure 2. In the following, we briefly list the most important features of the API.

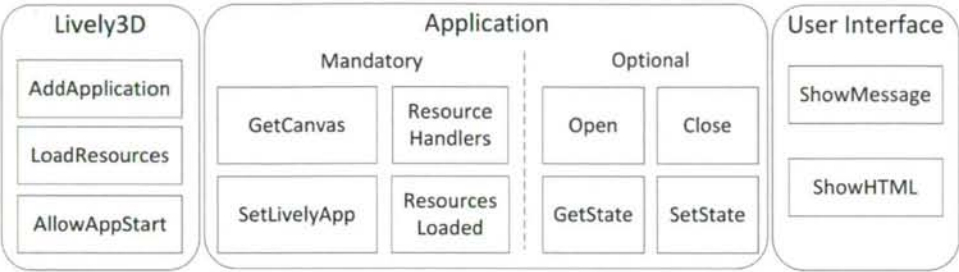


Figure 2: Lively3D API for applications.

To convert existing application to Lively3D app, the application must implement mandatory function of the figure. To embed the converted app to environment, the initialization code of the app must start the embedding process with calling the AddApplication-function. The process is presented in Figure 3.

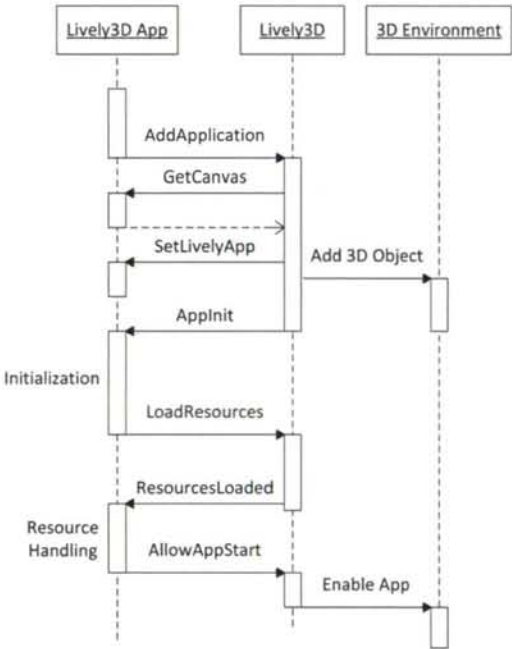


Figure 3: Sequence for embedding new Lively3D app.

As illustrated in the above figures, each application must implement a few mandatory functions and call Lively3D functions in certain order to advance the integration with the environment. During the integration, the canvas app is created and hidden with CSS-styling.

The Lively3D framework creates 3D objects representing the app and texturizes

them with the canvas element. Additionally to the mandatory functions, apps can provide optional functions which react to events like opening and closing the application within the environment. These functions have default functionality if they are unimplemented, but when the developer decides to provide them, they define what happens to the application status during the different events. Additionally, the inner state of the application can be serialized and de-serialized to developer's desired format.

Since the canvas element is defined as the only graphical element allowed for Lively3D Apps, the API also provides user interface functions to display messages and HTML in Lively3D provided dialogs. This provides consistent user interface, since Lively3D itself is rendered in a full browser window and possibilities of displaying text or other web interface elements within the environment are limited due to the WebGL specification. Figure 4 illustrates the existing canvas application in the left and the conversion to Lively3D app in the right with another app in the same environment.

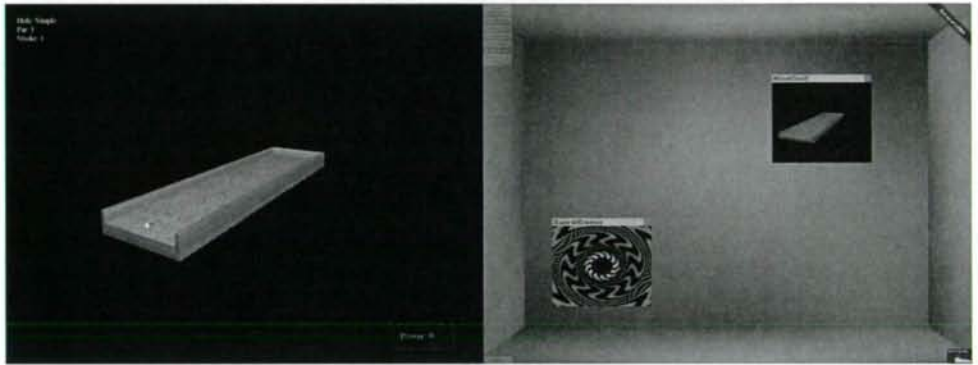


Figure 4: Conversion of existing application.

3.3 Redefining the 3D environment

As is common in various 3D applications, including in particular the genre of computer games, the visualization in our system is based on so-called scene graph, a generic tree-like data structure containing a collection of nodes. Nodes in the scene graph may have many children but most often they only need a single parent. In this structure, any operation performed to the parent is further propagated to its children. This flexible data structure enables numerous different visualizations, where the parent-children role can be benefited from.

The 3D environments in Lively3D are implemented dynamically, so that user can load new environments and change between them at will. As default only one environment is initialized in Lively3D and after adding more environments, the process of switching between environments is presented in Figure 5. Closing the applications and rebinding the events is done, so that the environment is in known

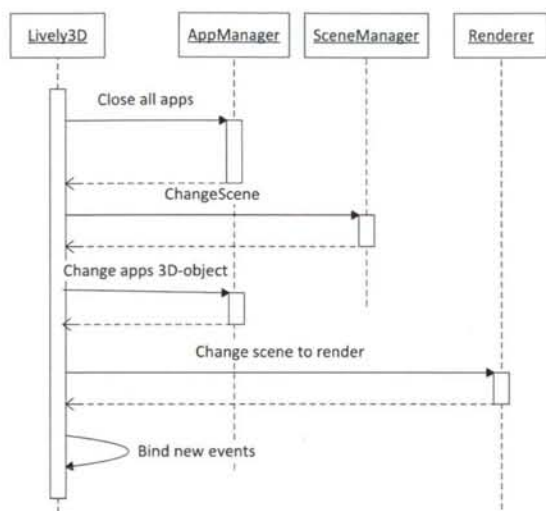


Figure 5: Sequence of switching environment.

initial state. Changing of the 3D-objects is required since GLGE allows 3D-object to be present only in one scene at a time.

In our experiment, we have created three different ways to visualize a scene graph where the children are applications and the root node is the 3D environment hosting the children. Example host environments include a conventional desktop, a planetary system where applications rotate a sun like in a solar system, and a true 3D virtual world, where applications move in a 3D terrain. These are introduced in the following in more detail, together with a set of screen shots to demonstrate their visual appearance.

Desktop. The conventional desktop consists of three dimensional room, cubes that represent closed applications, and planes that act as individual applications,

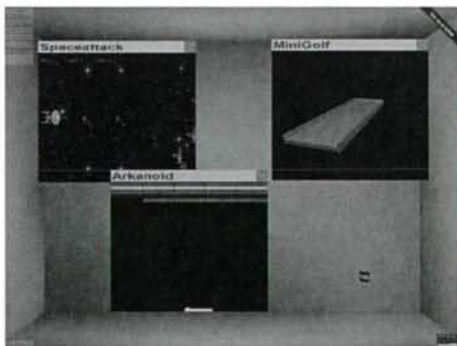


Figure 6: Visualizing the system as a conventional desktop.

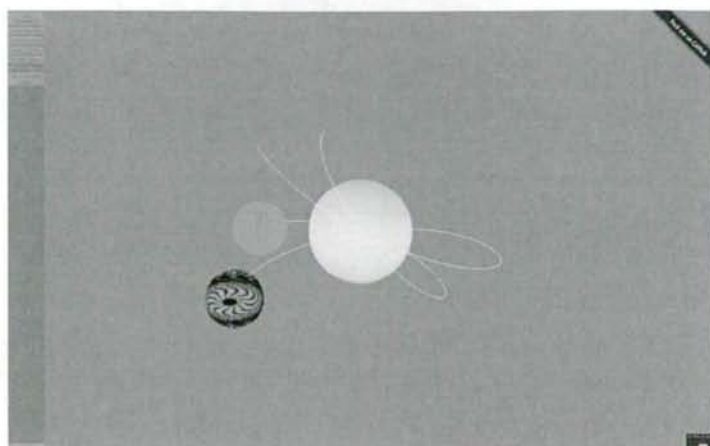


Figure 7: Visualizing the system as a solar system.

with the ability to execute JavaScript code, render to the screen, and so forth. A screenshot of the desktop environment, with three opened and two closed applications, is presented in Figure 6. The scene mimics all traditional desktop features, including dragging applications within the desktop and application interaction with opening, closing, maximizing and minimizing them with mouse controls.

Solar system. The solar system scene modifies the presentation of applications. In this scene, applications are presented as spheres that revolve around the central sun. Each revolving sphere generates a white trace in accordance to its path, and the trace is removed when the trace reaches maximum length. Each sphere uses the texture of the application canvas it is representing, and therefore each sphere has a different look within the scene. An example scene with 4 applications is demonstrated in Figure 7. Application windows retain their default functionality with dragging around, maximizing, minimizing, and so on. When an application that has been moved around is closed, the application returns to its position revolving around the central sun, in comparison to the conventional desktop scene where the application simply retains its current position.

Virtual world. The 3D virtual world scene goes even further from the conventional desktop. The only thing retained from the desktop concept are the application windows, and the only remaining controls for the windows are opening and closing the application, which then of course can introduce more controls within the application. The world itself consists of three dimensional terrain, where the user can wander around using the keyboard and the mouse. In this setting, applications are presented as spheres that roam the terrain in random directions, with their textures simplified to single image for performance reasons - experiences where application textures were used quickly showed that the resources of the test computer would no longer be adequate for such cases. Using this visualization, the 3D terrain and seven sample application spheres are illustrated in Figure 8. The right side of



Figure 8: Visualizing the system as a 3D virtual world.

the figure illustrates application canvases within the world.

All of the above visualizations are based on the same JavaScript code, with the only difference being the rendering strategy associated with the scene graph. Consequently, in all of these systems applications are runnable, and can in fact run even when they are inactive and being managed by the different host environments, except when explicitly disabled for performance reasons.

4 Refactoring Lively3D UI

In this section, we introduce some early experiences regarding the relation between the Lively3D framework and widget libraries commonly used in desktop applications. To summarize problems, the original implementation was built directly on primitives emerging from WebGL, whereas the refactored version is geared towards widget libraries in its architecture.

4.1 Identified Problems

As a part of the process of designing the Lively3D framework, it became obvious that its architecture would benefit from more abstract programming concepts, in particular when considering the programming of the 3D UI. WebGL is a low abstraction level tool and 3D-engines building upon it only hide the rendering details from programmer. In particular such libraries lack essential concepts known from desktop application development.

As a concrete example, let us examine Lively3D's application window. The application window is a composition of three different 3D objects – title bar, window content and close button. These 3D objects are grouped together and aligned so that they appear as a window that is a solid object.

The background is that WebGL provides tools to create the 3D objects, align and group those, but there is no tools for creating a WIMP¹⁸ elements such as titled window which can be dragged from the title bar and closed from the close

¹⁸Windows, Icons, Menus, and Pointer

button. However the natural abstraction of application window is an UI widget which has predefined look and feel, not a group of geometries which application logic is responsible for, which was the case in our original implementation.

Most of the 3D engines built for WebGL lack also necessary event handling capabilities. Using e.g. GLGE there is no way to bind an event listener to a 3D object. Determining which event happened and which object receives the event is responsibility of an application developer. In Lively3D the event handling for 3D UI is mixed into Lively3D application logic. In Lively3D there is a main event handler which catches all events for the Lively3D canvas, determines which object receives the event and executes functionality related to that object.

For instance, if the user clicks the close button of a window, Lively3D's main event handler will receive the event and calculate collision detection based on the mouse position to determine if the mouse hit any 3D objects. After finding the 3D object the event handler has to deduce which kind of object was hit and execute operations related to that object. In the window's close buttons case the operation would be to hide the group of 3D objects that forms the window.

In Lively3D there are only two kinds of 3D widgets – application windows and application icons – which receives only restricted amount of events so Lively3D has fairly simple 3D UI. However if we wish to add some new interactive 3D content to Lively3D we would need to refactor quite a lot of Lively3D code to get that done. Simple 3D UIs can be built using low abstraction level tools however the UI definition and logic becomes easily a mess of glut and glue solutions which makes it hard to maintain and develop the application further.

4.2 Revisiting the Design

Motivated by the above observations we created WebWidget3D, a 3D widget library for WebGL [3]. The idea of the library is to provide some predefined reusable 3D widgets and tools for building custom 3D widgets. WebWidget3D provides event system which enables binding mouse and keyboard events directly to 3D widgets. The framework also introduces predefined controls e.g. drag control and roll control which can be bind to any widget and fly control for moving camera in the 3D scene. The current implementation of WebWidget3D uses Three.js 3D engine for rendering, although 3D engine can be changed due specialized adapter component.

WebWidget3D provides predefined widgets and abstraction for creating widgets but it does not force the 3D world to consist of only 3D widgets. WebWidget3D content can be mixed with content (e.g. 3D objects, visual effects, animations, physics, etc.) provided by the 3D engine used with WebWidget3D.

We redesigned and reimplemented Lively3D's desktop UI using WebWidget3D to see how much refactoring would affect to Lively3D's complexity. The implementation is divided to two parts, 1) widget building blocks out of which complete widgets can be built (Table 1), and 2) a reduced set of ready-to-use widgets that can be used to create complete applications (Table 2). Figure 9 illustrates the revisited implementation.

Table 1: Building blocks of revised Lively3D design

Component	Description
GuiObject	Basic event handling capabilities.
Widget	Numerous commonly needed facilities for creating applications. Base class for new widgets.
Text	Simple string handling functionality.
Group	Abstraction of a container that can have other components as its children. Container can also have a 3D object representation.
Application	Corresponds to an application; receives events.

Table 2: Widgets used in Lively3D

Widget	Description
Grid window	Instance of a Group that is represented as a 3D grid plane. The grid window widget can be rotated in 3D space with the mouse.
Titled window	Instance of class Group. Contains three instances of Widget class as a title bar, a close button, and for representing the window content.
Menu window	Menu composed of multiple choice buttons. Individual choices are represented as cuboids.
Dialog window	Dialog composed of title text, multiple text fields and multiple action buttons.

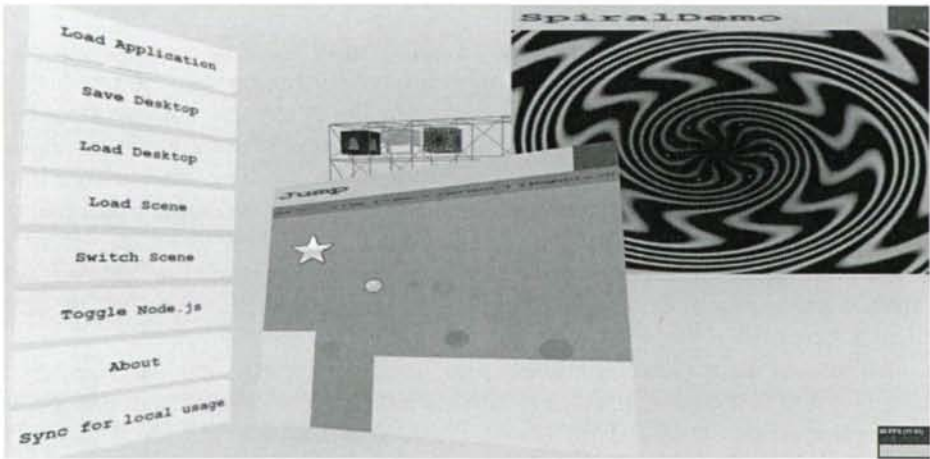


Figure 9: Reimplemented Lively3D.

4.3 Evaluation

In general, our work supports the conclusions of [2], where architecture related aspects of traditional applications are used as a driver for the design of apps that re run inside web pages. The goal of such designs, commonly referred to as single page web applications, is to support reuse and modifications in the long run, thus sharing the goals of more traditional software systems. Above, we reuse the design paradigm that is mature in the field of desktops, but to some extent missing from web design.

To evaluate the design, using WebWidget3D and its predefined widgets we were able to reduce the amount of JavaScript code lines by 26% [3]. In addition, using the library liberates developers to focus on solving application specific problems by allowing them to overlook numerous details that remain similar in different applications.

We also replaced Lively3D's 2D UI (menus and dialogs) with corresponding 3D UI widgets. This design reduced the number of code lines of HTML and CSS but on the other hand increased lines of code of JavaScript code.

5 Conclusions

Lively3D framework presents architecture to download and execute different applications within same environment. Although similar windowing environments have been developed and studied for years like Compiz/Beryl¹⁹, our experiment runs on top of the browser. This approach has the advantage of the cloud, so that the user does not need to install anything else except the browser to execute the environment and the applications. This approach also works in different platforms from desktop Windows and Linux to mobile phones.

Our prototype demonstrates that integrating individual applications in a single web page is possible and achievable without complex structures from the application developer. However, one of the main goals – using existing content, preferably complete web sites in the system as applications – turned out to be unreachable. Due to the WebGL specification limitations, the use of existing content as textures is limited to image, video, and canvas elements, whereas in order to render existing web pages within 3D environment, the WebGL specification should support IFrames as a source for textures. Currently, this option is associated with security issues - using the WebGL API gives loaded applications a direct access to the host devices hardware - which must be resolved before extending the rendering capabilities. Until then, applications are limited to the functionality of canvas element to produce graphics.

Additional security issues also emerge. Applications share the same JavaScript namespace which causes problems with variable overwriting. Even though each application has a simulated private namespace, variables might bleed through to the global namespace if the variable is missing var keyword. Applications can ac-

¹⁹<http://www.compiz.org>

cess global variables and overwrite them, including Lively3D namespace, other used JavaScript libraries and even browsers' default JavaScript functionality. This especially causes accidental problems with generic JavaScript libraries, since they are usually bound in `$` variable, which is overwritten when new library is loaded and basic functionality of the environment brakes down as result. These problems could be fixed with proper process model where each application has its own private namespace and rendering context. There has been an emergence of JavaScript frameworks like `Require.js`²⁰ and `browserify`²¹ that encapsulate parts of the JavaScript code to separate modules, this could be used as a pattern to fix some of the problems of Lively3D.

The Single Page Application paradigm has its advantages and disadvantages. Even though applications are in the same JavaScript namespace, this could be leveraged so that applications could communicate with each other. To enable this, the environment would need common JavaScript interfaces for application communications. Current implementation does not provide documented APIs for this.

One of the goals of Lively3D was minimal overhead code while embedding existing applications. We consider that this requirement was achieved quite well, although comprehensive analysis between converted applications is useless since amount of overhead code depends on coding conventions. In Lively3D most of the application initialization must be done dynamically in JavaScript code, as opposed to convential browser where HTML tags can handle some of the resource downloading. The minimal overhead code amounts to about 50 lines of extra code.

In the course of the design, we were alarmed by the fact that the circumvention of security restrictions became one of the key design drivers in the experiment. In this field, the problems arise from the combination of the current "one size fits all" browser security model and the general document-oriented nature of the web browser. Decisions about security are determined primarily by the site (origin) from which the web document is loaded, not by the specific needs of the document or application. Such problems could be alleviated by introducing a more fine-grained security model, e.g., a model similar to the comprehensive security model of the Java SE platform [1] or the more lightweight, permission-based, certificate-based security model introduced by the MIDP 2.0 Specification for the Java Platform, Micro Edition (Java ME) [4]. As already pointed out in [6], the biggest challenges in this area are related to standardization, as it is difficult to define a security solution that would be satisfactory to everybody while retaining backwards compatibility.

Finally, there are numerous new methodological issues associated with the transition. The transition from conventional applications to web applications will result in a shift away from static programming languages such as C, C++ or C# towards dynamic programming languages. Since mainstream software developers are often unaware of the fundamental development style differences between static and dynamic programming languages, they need to be educated about the evolutionary, exploratory programming style associated with dynamic languages. Furthermore,

²⁰<http://requirejs.org/>

²¹<http://browserify.org/>

techniques associated with dealing with big data – data sets that are too large to work with using on-hand database management tools – data mining, and mashup development will be increasingly important.

To conclude, when considering the humble beginnings of the web browser as a simple document viewing and distribution environment, and the fact that programmatic capabilities on the Web were largely an afterthought rather than a carefully designed feature, the transformation of the Web into an extremely popular software deployment platform is amazing. This transformation is one of the most profound changes in the modern history of computing and software engineering.

In this paper, we are demonstrating the effect of new ways to visualize content in a fashion where the browser's new extensions are based on new web protocols rather than plugins, which has been the traditional way to create richer media inside the browser. Since no plugins that commonly introduce restrictions associated with their proprietary origins, the new technologies are manifesting the open web and open data. This, together with open data that is be available to everyone to freely use and republish as they wish without mechanisms of control, in turn liberates the developers to create increasingly compelling applications, building on the facilities that already exist in the web as well as their own innovative ideas.

References

- [1] Gong, Li and Ellison, Gary. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2nd edition, 2003.
- [2] Kuuskeri, Janne. *Engineering web applications: Architectural principles for web software*. Tampere University of Technology, 2014.
- [3] Mattila, Anna-Liisa and Mikkonen, Tommi. Designing a 3d widget library for WebGL enabled browsers. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 757–760, New York, NY, USA, 2013. ACM.
- [4] Riggs, Roger, Huopaniemi, Jyri, Taivalsaari, Antero, Patel, Mark, and Uotila, Aleks. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Sun Microsystems, Inc., Mountain View, CA, USA, 2 edition, 2003.
- [5] Taivalsaari, Antero, Mikkonen, Tommi, Anttonen, Matti, and Salminen, Arto. The death of binary software: End user software moves to the web. In *Proceedings of the 2011 Ninth International Conference on Creating, Connecting and Collaborating Through Computing, C5 '11*, pages 17–23, Washington, DC, USA, 2011. IEEE Computer Society.
- [6] Taivalsaari, Antero, Mikkonen, Tommi, Ingalls, Dan, and Palacz, Krzysztof. Web browser as an application platform. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications, SEAA '08*, pages 293–302, Washington, DC, USA, 2008. IEEE Computer Society.

Asymptotic Proportion of Hard Instances of the Halting Problem

Antti Valmari*

Abstract

Although the halting problem is undecidable, imperfect testers that fail on some instances are possible. Such instances are called *hard* for the tester. One variant of imperfect testers replies “I don’t know” on hard instances, another variant fails to halt, and yet another replies incorrectly “yes” or “no”. Also the halting problem has three variants: does a given program halt on the empty input, does a given program halt when given itself as its input, or does a given program halt on a given input. The failure rate of a tester for some size is the proportion of hard instances among all instances of that size. This publication investigates the behaviour of the failure rate as the size grows without limit. Earlier results are surveyed and new results are proven. Some of them use C++ on Linux as the computational model. It turns out that the behaviour is sensitive to the details of the programming language or computational model, but in many cases it is possible to prove that the proportion of hard instances does not vanish.

Keywords: halting problem, three-way tester, generic-case tester, approximating tester

1 Introduction

Turing proved in 1936 that undecidability exists by showing that the halting problem is undecidable [10]. Rice extended the set of known undecidable problems to cover all questions of the form “does the partial function computed by the given program have property X ”, where X is any property that at least one computable partial function has and at least one does not have [7]. For instance, X could be “returns 1 for all syntactically correct C++ programs and 0 for all remaining inputs.” In other words, it may be impossible to find out whether a given weird-looking program is a correct C++ syntax checker. These results are basic material in such textbooks as [3].

On the other hand, imperfect halting testers are possible. For any instance of the halting problem, a *three-way tester* eventually answers “yes”, “no”, or “I

*Tampere University of Technology, Department of Mathematics, PO Box 553, FI-33101 Tampere, FINLAND, E-mail: Antti.Valmari@tut.fi

don't know". If it answers "yes" or "no", then it must be correct. We say that the "I don't know" instances are *hard instances* for the tester. Also other kinds of imperfect testers have been introduced, as will be discussed in Section 2.1.

Assume that T_1 is a tester. By Turing's proof, it has a hard instance I_1 . If I_1 is a halting instance, then let T_2 be "if the input is I_1 , then reply 'yes', otherwise run T_1 and return its reply". If I_1 is non-halting, then let T_2 be "if the input is I_1 , then reply 'no', otherwise run T_1 and return its reply". By construction, T_2 is a tester with one fewer hard instances than T_1 has. By Turing's proof, also T_2 has a hard instance. Let us call it I_2 . It is hard also for T_1 . This reasoning can be repeated without limit, yielding an infinite sequence T_1, T_2, \dots of testers and I_1, I_2, \dots of instances such that I_i is hard for T_1, \dots, T_i but not for T_{i+1}, \dots . Therefore, every tester has an infinite number of hard instances, but no instance is hard for all testers.

A program that answers "I don't know" for every program and input is a three-way tester, although it is useless. A much more careful tester simulates the given program on the given input at most 9^n steps, where n is the joint size of the program and its input. If the program stops by then, then the tester answers "yes". If the program repeats a configuration (that is, a complete description of the values of variables, the program counter, etc.) by then, then the tester answers "no". Otherwise it answers "I don't know". With this theoretically possible but in practice unrealistic tester, any hard halting instance has a finite but very long running time.

The proofs by Turing and Rice may leave the hope that only rare artificial contrived programs yield hard instances. One could dream of a three-way tester that answers very seldom "I don't know". This publication analyses this issue, by surveying and proving results that tell how the proportion of hard instances behaves when the size of the instances grows without limit.

Section 2 presents the variants of the halting problem and imperfect testers surveyed, together with some basic results and notation. Earlier research is discussed in Section 3. The section contains some proofs to bring results into the framework of this publication. Section 4 presents some new results in the case that a program has many copies of all big sizes, or information can be packed densely inside the program. It is not always assumed that the program has access to the information. A natural example of such information is dead code, such as `if(1==0)then{...}`. In Section 5, results are derived for C++ programs with inputs from files. Section 6 briefly concludes this publication.

This publication is a significantly extended version of [12, 13]. The papers [12, 13] are otherwise essentially the same, but three proofs were left out from [13] because of lack of space. In the present publication, Theorems 4 and 6 and Corollaries 2 and 4 are new results lacking from [12, 13]. Furthermore, [12, 13] incorrectly claimed the opposite of Theorem 6. The present publication fixes this error and also a small error in Proposition 4.

2 Concepts and Notation

2.1 Variants of the Halting Problem

The literature on hard instances of the halting problem considers at least three variants of the halting problem:

E does the given program halt on the *empty* input [2],

S does the given program halt when given *itself* as its input [6, 8], and

G does the given program halt on the *given* input [1, 4, 9].

Each variant is undecidable. Variant G has a different notion of instances from others: program–input pairs instead of just programs. A tester for G can be trivially converted to a tester for E or S, but the proportion of hard program–input pairs among all program–input pairs of some size is not necessarily the same as the similar proportion with the input fixed to the empty one or to the program itself.

The literature also varies on what the tester does when it fails. Three-way testers, that is, the “I don’t know” answer is used implicitly by [6], as it discusses the union of two decidable sets, one being a subset of the halting and the other of the non-halting instances. In *generic-case decidability* [8], instead of the “I don’t know” answer, the tester itself fails to halt. Yet another idea is to always give a “yes” or “no” answer, but let the answer be incorrect for some instances [4, 9]. Such a tester is called *approximating*. One-sided results, where the answer is either “yes” or “I don’t know”, were presented in [1, 2]. For a tester of any of the three variants, we say that an instance is *easy* if the tester correctly answers “yes” or “no” on it, otherwise the instance is *hard*.

These yield altogether nine different sets of testers, which we will denote with three-way(X), generic(X), and approx(X), where X is E, S, or G. Some simple facts facilitate carrying some results from one variant of testers to another.

Proposition 1. *For any three-way tester there is a generic-case tester that has precisely the same easy “yes”-instances, easy “no”-instances, hard halting instances, and hard non-halting instances.*

There also is an approximating tester that has precisely the same easy “yes”-instances, at least the same easy “no”-instances, precisely the same hard halting instances, and no hard non-halting instances; and an approximating tester that has at least the same easy “yes”-instances, precisely the same easy “no”-instances, no hard halting instances, and precisely the same hard non-halting instances.

Proof. A three-way tester can be trivially converted to the promised tester by replacing the “I don’t know” answer with an eternal loop, the reply “no”, or the reply “yes”. □

Proposition 2. *For any generic-case tester there is a generic-case tester that has at least the same “yes”-instances, precisely the same “no”-instances, no hard halting instances, and precisely the same hard non-halting instances.*

Proof. In parallel with the original tester, the instance is simulated. (In Turing machine terminology, parallel simulation is called “dovetailing”.) If the original tester replies something, the simulation is aborted. If the simulation halts, the original tester is aborted and the reply “yes” is returned. \square

Proposition 3. *For any $i \in \mathbb{N}$ and tester T , there is a tester T_i that answers correctly “yes” or “no” for all instances of size at most i , and similarly to T for bigger instances.*

Proof. Because there are only finitely many instances of size at most i , there is a finite bit string that lists the correct answers for them. If $n \leq i$, T_i picks the answer from it and otherwise calls T . (We do not necessarily know what bit string is the right one, but that does not rule out its existence.) \square

2.2 Notation

We use Σ to denote the set of characters that are used for writing programs and their inputs. It is finite and has at least two elements. There are $|\Sigma|^n$ character strings of size n . If α and β are in Σ^* , then $\alpha \sqsubseteq \beta$ denotes that α is a prefix of β , and $\alpha \subset \beta$ denotes proper prefix. The size of α is denoted with $|\alpha|$.

A set A of finite character strings is *self-delimiting* if and only if membership in A is decidable and no member of A is a proper prefix of a member of A . The *shortlex ordering* of any set of finite character strings is obtained by sorting the strings in the set primarily according to their sizes and strings of the same size in the lexicographic order.

Not necessarily all elements of Σ^* are programs. The set of programs is denoted with Π , and the set of all (not necessarily proper) prefixes of programs with Γ . So $\Pi \subseteq \Gamma$. For tester variants E and S, we use $p(n)$ to denote the number of programs of size n . Then $p(n) = |\Sigma^n \cap \Pi|$. For tester variant G, $p(n)$ denotes the number of program-input pairs of joint size n . We will later discuss how the program and its input are paired into a single string. The numbers of halting and non-halting (a.k.a. diverging) instances of size n are denoted with $h(n)$ and $d(n)$, respectively. We have $p(n) = h(n) + d(n)$.

If T is a tester, then $\underline{h}_T(n)$, $\bar{h}_T(n)$, $\underline{d}_T(n)$, and $\bar{d}_T(n)$ denote the number of its easy halting, hard halting, easy non-halting, and hard non-halting instances of size n , respectively. Obviously $\underline{h}_T(n) + \bar{h}_T(n) = h(n)$ and $\underline{d}_T(n) + \bar{d}_T(n) = d(n)$. The smaller $\bar{h}_T(n)$ and $\bar{d}_T(n)$ are, the better the tester is. The *failure rate* of T is $(\bar{h}_T(n) + \bar{d}_T(n))/p(n)$.

When referring to all instances of size at most n , we use capital letters. So, for example, $P(n) = \sum_{i=0}^n p(i)$ and $\bar{D}_T(n) = \sum_{i=0}^n \bar{d}_T(i)$.

3 Related Work

3.1 Early Results by Lynch

Nancy Lynch [6] used *Gödel numberings* for discussing programs. In essence, it means that each program has at least one index number (which is a natural number) from which the program can be constructed, and each natural number is the index of some program.

Although the index of an individual program may be smaller than the index of some shorter program, the overall trend is that indices grow as the size of the programs grows, because otherwise we would run out of small numbers. On the other hand, if the mapping between the programs and indices is 1-1, then the growth cannot be faster than exponential. This is because $p(n) \leq |\Sigma|^n$. With real-life programming languages, the growth is exponential, but (as we will see in Section 5.2) the base of the exponent may be smaller than $|\Sigma|$.

To avoid confusion, we refrain from using the notation \overline{H}_T , etc., when discussing results in [6], because the results use indices instead of sizes of programs, and their relationship is not entirely straightforward. Fortunately, some results of [6] can be immediately applied to programming languages by using the *shortlex Gödel numbering*. The shortlex Gödel number of a program is its index in the shortlex ordering of all programs.

The first group of results of [6] reveals that a wide variety of situations may be obtained by spreading the indices of all programs sparsely enough and then filling the gaps in a suitable way. For instance, with one Gödel numbering, for each three-way tester, the proportion of hard instances among the first i indices approaches 1 as i grows. With another Gödel numbering, there is a three-way tester such that the proportion approaches 0 as i grows. There even is a Gödel numbering such that as i grows, the proportion oscillates in the following sense: for some three-way tester, it comes arbitrarily close to 0 infinitely often and for each three-way tester, it comes arbitrarily close to 1 infinitely often.

In its simplest form, spreading the indices is analogous to defining a new language SpaciousC++ whose syntax is identical to that of C++ but the semantics is different. If the first $\lfloor n/2 \rfloor$ characters of a SpaciousC++ program of size n are space characters, then the program is executed like a C++ program, otherwise it halts immediately. This does not restrict the expressiveness of the language, because any C++ program can be converted to a similarly behaving SpaciousC++ program by adding sufficiently many space characters to its front. However, it makes the proportion of easily recognizable trivially halting instances overwhelm. A program that replies “yes” if there are fewer than $\lfloor n/2 \rfloor$ space characters at the front and “I don’t know” otherwise, is a three-way tester. Its proportion of hard instances vanishes as the size of the program grows.

As a consequence of this and Proposition 3, one may choose any failure rate above zero and there is a three-way tester for SpaciousC++ programs with at most that failure rate. Of course, this result does not tell anything about how hard it is to test the halting of interesting programs. This is the first example in this

publication of what we call *an anomaly stealing the result*. That is, a proof of a theorem goes through for a reason that has little to do with the phenomenon we are interested in.

Indeed, the first results of [6] depend on using unnatural Gödel numberings. They do not tell what happens with untampered programming languages. Even so, they rule out the possibility of a simple and powerful general theorem that applies to all models of computation. They also make it necessary to be careful with the assumptions that are made about the programming language.

To get sharper results, *optimal Gödel numberings* were discussed in [6]. They do not allow distributing programs arbitrarily. A Gödel numbering is optimal if and only if for any Gödel numbering, there is a computable function that maps it to the former such that the index never grows more than by a constant factor.¹ The most interesting sharper results are opposite to what was obtained without the optimality assumption. To apply them to programming languages, we first define a programming language version of optimal Gödel numberings.

Definition 1. *A programming language is end-of-file data segment, if and only if each program consists of two parts in the following way. The first part, called the actual program, is written in a self-delimiting language (so its end can be detected). The second part, called the data segment, is an arbitrary character string that extends to the end of the file. The language has a construct via which the actual program can read the contents of the data segment.*

The data segment is thus a data literal in the program, packed with maximum density. It is not the same thing as the input to the program.

Corollary 1. *For each end-of-file data segment language,*

$$\exists c > 0 : \exists T \in \text{three-way}(S) : \forall n \in \mathbb{N} : \frac{H_T(n) + D_T(n)}{P(n)} \geq c \text{ and}$$

$$\exists c > 0 : \forall T \in \text{three-way}(S) : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\overline{H}_T(n) + \overline{D}_T(n)}{P(n)} \geq c .$$

Proof. Let \mathcal{L} be the end-of-file data segment language, and let \mathcal{G} be any Gödel numbering. Consider the following program P in \mathcal{L} . Let a and d be the sizes of its actual program and data segment. The actual program reads the data segment, interpreting its content as a number i in the range from $\frac{|\Sigma|^d - 1}{|\Sigma| - 1} + 1$ to $\frac{|\Sigma|^{d+1} - 1}{|\Sigma| - 1}$. Then it simulates the i th program in \mathcal{G} . The shortlex index of P is at most $i' = \sum_{j=0}^{a+d} |\Sigma|^j \leq |\Sigma|^{a+d+1}$. We have $\frac{|\Sigma|^d - 1}{|\Sigma| - 1} + 1 \leq i$, yielding $|\Sigma|^d - 1 \leq |\Sigma|i - i - |\Sigma| + 1$, so $|\Sigma|^d \leq |\Sigma|i$, thus $i' \leq |\Sigma|^{a+2}i$. The shortlex numbering of \mathcal{L} is thus an optimal Gödel numbering. From this, Proposition 6 in [6] gives the claims. \square

¹The definition in [6] seems to say that the function must be a bijection. We believe that this is a misprint, because each proof in [6] that uses optimal Gödel numberings obviously violates it.

A remarkable feature of the latter result compared to many others in this publication is that c is chosen before T . That is, there is a positive constant that only depends on the programming language (and not on the choice of the tester) such that all testers have at least that proportion of hard instances, for any big enough n . On the other hand, the proof depends on the programming language allowing to pack raw data very densely. Real-life programming languages do not satisfy this assumption. For instance, C++ string literals "... " cannot pack data densely enough, because the representation of " inside the literal (e.g., \ " or \042) requires more than one character.

Because of Proposition 3, " $\exists n_T \in \mathbb{N}$ " cannot be moved to the front of " $\forall T \in$ three-way(S)".

The result cannot be generalized to \bar{h}_T , \bar{d}_T , and p , because the following anomaly steals it. We can change the language by first adding 1 or 01 to the beginning of each program π and then declaring that if the size of 1π or 01π is odd, then it halts immediately, otherwise it behaves like π . This trick does not invalidate optimality but introduces infinitely many sizes for which the proportion of hard instances is 0.

3.2 Results on Domain-Frequent Programming Languages

In [4], the halting problem was analyzed in the context of programming languages that are *frequent* in the following sense:

Definition 2. A programming language is (a) frequent (b) domain-frequent, if and only if for every program π , there are $n_\pi \in \mathbb{N}$ and $c_\pi > 0$ such that for every $n \geq n_\pi$, at least $c_\pi p(n)$ programs of size n (a) compute the same partial function as π (b) halt on precisely the same inputs as π .

Instead of "frequent", the word "dense" was used in [4], but we renamed the concept because we felt "dense" a bit misleading. The definition says that programs that compute the same partial function are common. However, the more common they are, the less room there is for programs that compute other partial functions, implying that the smallest programs for each distinct partial function must be distributed more sparsely. "Dense" was used for domain-frequent in [9].

Any frequent programming language is obviously domain-frequent but not necessarily vice versa. On the other hand, even if a theorem in this field mentions frequency as an assumption, the odds are that its proof goes through with domain-frequency. Whether a real-life programming language such as C++ is domain-frequent, is surprisingly difficult to find out. We will discuss this question briefly in Section 4.1.

As an example of a frequent programming language, BF was mentioned in [4]. Its full name starts with "brain" and then contains a word that is widely considered inappropriate language, so we follow the convention of [4] and call it BF. Information on it can be found on Wikipedia under its real name. It is an exceptionally simple programming language suitable for recreational and illustrational but not for real-life programming purposes. In essence, BF programs describe Turing machines with

a read-only input tape, write-only output tape, and one work tape. The alphabet of each tape is the set of 8-bit bytes. However, BF programs only use eight characters.

As a side issue, a non-trivial proof was given in [4] that only a vanishing proportion of character strings over the eight characters are BF programs. That is, $\lim_{n \rightarrow \infty} p(n)/8^n$ exists and is 0. It trivially follows that if all character strings over the 8 characters are considered as instances and failure to compile is considered as non-halting, then the proportion of hard instances vanishes as n grows.

The only possible compile-time error in BF is that the square brackets [and] do not match. Most, if not all, real-life programming languages have parentheses or brackets that must match. So it seems likely that compile-time errors dominate also in the case of most, if not all, real-life programming languages. Unfortunately, this is difficult to check rigorously, because the syntax and other compile-time rules of real-life programming languages are complicated. Using another, simpler line of argument, we will prove the result for both C++ and BF in Section 5.1.

In any event, if the proportion of hard instances among all character strings vanishes because the proportion of programs vanishes, that is yet another example of an anomaly stealing the result. It is uninteresting in itself, but it rules out the possibility of interesting results about the proportion of hard instances of size n among all character strings of size n . Therefore, from now on, excluding Section 5.1, we focus on the proportion of hard instances among all programs or program-input pairs.

In the case of program-input pairs, the results may be sensitive to how the program and its input are combined into a single string that is used as the input of the tester. To avoid anomalous results, it was assumed in [4, 9] that this “pairing function” has a certain property called “pair-fair”. The commonly used function $x + (x+y)(x+y+1)/2$ is pair-fair. To use this pairing function, strings are mapped to numbers and back via their indices in the shortlex ordering of all finite character strings.

A proof was sketched in [9] that, assuming domain-frequency and pair-fairness,

$$\forall T \in \text{approx}(G) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \geq c_T .$$

That is, the proportion of wrong answers does not vanish. However, this leaves open the possibility that for any failure rate $c > 0$, there is a tester that fares better than that for all big enough n . This possibility was ruled out in [4], assuming frequency and pair-fairness. (It is probably not important that frequency instead of domain-frequency was assumed.) That is, there is a positive constant such that for any tester, the proportion of wrong answers exceeds the constant for infinitely many sizes of instances:

$$\exists c > 0 : \forall T \in \text{approx}(G) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \geq c . \quad (1)$$

The third main result in [4], adapted and generalized to the present setting, is the following. We present its proof to obtain the generalization and to add a detail

that the proof in [4] lacks, that is, how $T_{i,j}$ is made to halt for “wrong sizes”. Generic-case testers are not mentioned, because Proposition 2 gave a related result for them.

Theorem 1. *For each programming model and variant E, S, G of the halting problem,*

$$\forall c > 0 : \exists T_c \in \text{approx}(X) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_{T_c}(n)}{p(n)} \leq c \wedge \frac{\bar{d}_{T_c}(n)}{p(n)} = 0 \text{ and}$$

$$\forall c > 0 : \exists T_c \in \text{three-way}(X) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_{T_c}(n)}{p(n)} \leq c .$$

Proof. Let $C = \lceil 1/c \rceil$. Consider the family $T_{i,j}$ of the programs of the following kind, where $i \in \mathbb{N}$, $j \in \mathbb{N}$, and $0 \leq i \leq C$. If $n < j$, $T_{i,j}$ answers “no” in the case of approximating and “I don’t know” in the case of three-way testers. If $n \geq j$, $T_{i,j}$ simulates all instances of size n until $\lceil ip(n)/C \rceil$ of them have halted. If the simulation stage terminates, then if the given instance is among those that halted, $T_{i,j}$ answers “yes”, otherwise $T_{i,j}$ answers “no” or “I don’t know”. Thus an approximating $T_{i,j}$ has $\bar{d}_{T_{i,j}}(n) = 0$.

We prove next that some $T_{i,j}$ is the required tester. Let $i_n = \lfloor Ch(n)/p(n) \rfloor$. Then $i_n p(n)/C \leq h(n) < (i_n + 1)p(n)/C$. When $n \geq j$, the simulation stage of $T_{i_n,j}$ terminates and the proportion of hard halting instances of $T_{i_n,j}$ is less than $1/C \leq c$. Some $0 \leq i \leq C$ is the i_n for infinitely many values of n . Furthermore, there is a smallest such i . We denote it with i' . There also is a j such that when $n \geq j$, then $i_n \geq i'$. With these choices, $T_{i',j}$ always halts. \square

For a small enough c and the approximating tester T_c in Theorem 1, (1) implies that the failure rate of T_c oscillates, that is, does not approach any limit as $n \rightarrow \infty$. This observation is directly obtainable from Lemma 23 in [4].

3.3 Results on Turing Machines

For Turing machines with one-way infinite tape and randomly chosen transition function, the probability of falling off the left end of the tape before halting or repeating a state approaches 1 as the number of states grows [2]. The tester simulates the machine until it falls off the left end, halts, or repeats a state. If falling off the left end is considered as halting, then the proportion of hard instances vanishes as the size of the machine grows. This can be thought of as yet another example of an anomaly stealing the result.

Formally, $\exists T \in \text{three-way}(X) : \lim_{n \rightarrow \infty} (\bar{h}_T(n) + \bar{d}_T(n))/p(n) = 0$, that is,

$$\exists T \in \text{three-way}(X) : \forall c > 0 : \exists n_c \in \mathbb{N} : \forall n \geq n_c : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \leq c .$$

Here X may be E, S , or G . Although E was considered in [2], the proof also applies to S and G . Comparing the result to Theorem 2 in Section 4.1 reveals that

the representation of programs as transition functions of Turing machines is not domain-frequent.

On the other hand, independently of the tape model, the proportion does not vanish exponentially fast [8]. Like in [2], the proportion is computed on the transition functions, and not on some textual representations of the programs. The proof relies on the fact that any Turing machine has many obviously similarly behaving copies of bigger and bigger sizes. They are obtained by adding new states and transitions while keeping the original states and transitions intact. So the new states and transitions are unreachable. They are analogous to dead code. These copies are not common enough to satisfy Definition 2, but they are common enough to rule out exponentially fast vanishing. Generic-case decidability was used in [8], but the result applies also to three-way testers by Proposition 1.

The results in [1] are based on using weighted running times. For every positive integer k , the proportion of halting programs that do not halt within time $k + c$ is less than 2^{-k} , simply because the proportion of times greater than $k + c$ is less than 2^{-k} . The publication presents such a weighting that c is a computable constant.

Assume that programs are represented as self-delimiting bit strings on the input tape of a universal Turing machine. The smallest three-way tester of variant E that answers “yes” or “no” up to size n and “I don’t know” for bigger programs, is of size $n \pm O(1)$ [11].

4 Programming Languages with Assumptions

4.1 Domain-Frequent Languages

The assumption that the programming language is domain-frequent (Definition 2) makes it possible to use a small variation of the standard proof of the non-existence of halting testers, to prove that each halting tester of variant S has a non-vanishing set of hard instances. For three-way and generic-case testers, one can also say something about whether the hard instances are halting or not. Despite its simplicity, as far as we know, the following result has not been presented in the literature. However, see the comment on [9] in Section 3.2.

Theorem 2. *If the programming language is domain-frequent, then*

$$\forall T \in \text{three-way}(S) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{h}_T(n)}{p(n)} \geq c_T \wedge \frac{\bar{d}_T(n)}{p(n)} \geq c_T ,$$

$$\forall T \in \text{generic}(S) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{d}_T(n)}{p(n)} \geq c_T , \text{ and}$$

$$\forall T \in \text{approx}(S) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \geq c_T .$$

Proof. Let the execution of X with an input y be denoted with $X(y)$. For any T , consider the program P_T that first tries its input x with T . If $T(x)$ replies

“yes”, then $P_T(x)$ enters an eternal loop. If $T(x)$ replies “no”, then $P_T(x)$ halts immediately. The case that $T(x)$ replies “I don’t know” is discussed below. If $T(x)$ fails to halt, then $P_T(x)$ cannot continue and thus also fails to halt.

By the definition of domain-frequent, there are $c_T > 0$ and $n_T \in \mathbb{N}$ such that when $n \geq n_T$, at least $c_T p(n)$ programs halt on precisely the same inputs as P_T . Let P' be any such program. Consider $P_T(P')$. If $T(P')$ answers “yes”, then $P_T(P')$ fails to halt. Then also $P'(P')$ fails to halt. Thus “yes” cannot be the correct answer for $T(P')$. A similar reasoning reveals that also “no” cannot be the correct answer for $T(P')$. So P' is a hard instance for T .

Nothing more is needed to prove the claim for approximating testers. In the case of generic-case testers, the hard instances make T and thus P_T fail to halt, so they are non-halting instances.

In the case of three-way testers, all hard instances can be made halting instances by making P_T halt when T replies “I don’t know”. This proves the claim $\bar{h}_T(n)/p(n) \geq c_T$. The claim $\bar{d}_T(n)/p(n) \geq c_T$ is proven by making P_T enter an eternal loop when T replies “I don’t know”. These two proofs may yield different c_T values, but the smaller one of them is suitable for both. Similarly, the bigger of their n_T values is suitable for both. \square

The second claim of Theorem 2 lacks a $\bar{h}_T(n)$ part. Indeed, Proposition 2 says that with generic-case testers, $\bar{h}_T(n)$ can be made 0. With approximating testers, $\bar{h}_T(n)$ can be made 0 at the cost of $\bar{d}_T(n)$ becoming $d(n)$, by always replying “yes”. Similarly, $\bar{d}_T(n)$ can be made 0 by always replying “no”.

The next theorem applies to testers of variant E and presents some results similar to Theorem 2. To our knowledge, it is the first theorem of its kind that applies to the halting problem on the empty input. It assumes not only that many enough equivalent copies exist but also that they can be constructed. On the other hand, its equivalence only pays attention to the empty input.

Definition 3. *A programming language is computably empty-frequent if and only if there is a decidable equivalence relation “ \approx ” between programs such that*

- *for each program π , there are $c_\pi > 0$ and $n_\pi \in \mathbb{N}$ such that for every $n \geq n_\pi$, at least $c_\pi p(n)$ programs of size n are equivalent to π , and*
- *for each programs π and π' , if $\pi \approx \pi'$, then either both or none of π and π' halt on the empty input.*

If $\pi \approx \pi'$, we say that π' is a cousin of π .

It can be easily seen from [4] that BF is computably empty-frequent.

Theorem 3. *If the programming language is computably empty-frequent, then*

$$\forall T \in \text{three-way(E)} : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{d}_T(n)}{p(n)} \geq c_T .$$

The result also holds for generic-case testers but not for approximating testers.

Proof. Given any three-way tester T , consider a program P_T that behaves as follows. First it constructs its own code and stores it in a string variable. Hard-wiring the code of a program inside the program is somewhat tricky, but it is well known that it can be done. With Gödel numberings, the same can be obtained with Kleene's second recursion theorem.

Then P_T starts constructing its cousins of all sizes and tests each of them with T . By the assumption, there are $c_T > 0$ and $n_T \in \mathbb{N}$ such that for every $n \geq n_T$, P_T has at least $c_T p(n)$ cousins of size n . If T ever replies "yes", then P_T enters an eternal loop and thus does not continue testing its cousins. If T ever replies "no", then P_T halts immediately. If T replies "I don't know", then P_T tries the next cousin.

If T ever replies "yes", then P_T fails to halt on the empty input. By definition, also the tested cousin fails to halt on the empty input. So the answer "yes" would be incorrect. Similarly, if T ever replies "no", that would be incorrect. So T must reply "I don't know" for all cousins of P_T . They are thus hard instances for T . Because there are infinitely many of them, P_T does not halt, so they are non-halting.

To prove the result for generic-case testers, it suffices to run the tests of the cousins in parallel, that is, go around a loop where each test that has been started is executed one step and the next test is started. If any test ever replies "yes" or "no", P_T aborts all tests that it has started and then does the opposite of the reply.

A program that always replies "no" is an approximating tester with $\bar{d}_T(n) = 0$ for every $n \in \mathbb{N}$. \square

The results in this section and Section 3.2 motivate the question: are real-life programming languages domain-frequent? For instance, is C++ domain-frequent? Unfortunately, we have not been able to answer it. We try now to illustrate why it is difficult.

Given any C++ program, it is easy to construct many longer programs that behave in precisely the same way, by adding space characters, line feeds (denoted with $\backslash n$), comments, or dead code such as `if(0!=0){...}`. It is, however, hard to verify that many enough programs are obtained in this way. For instance, it might seem that many enough programs can be constructed with string literals. We now provide evidence that suggests (but does not prove) that it fails.

Any program of size n can be converted to $(|\Sigma| - 3)^k$ identically behaving programs of size $n + k + 12$ by adding `{char*s="σ";}` to the beginning of some function, where $\sigma \in (\Sigma \setminus \{", \backslash, \backslash n\})^k$. (The purpose of `{` and `}` is to hide the variable s , so that it does not collide with any other variable with the same name.) More programs are obtained by including escape codes such as `\"` to σ .

However, it seems that this is a vanishing instead of at least a positive constant proportion when $k \rightarrow \infty$. In the absence of escape codes, it certainly is a vanishing proportion. This is because one can add `{char*s="σ",*t="ρ";}` instead, where $|\sigma| + |\rho| = k - 6$. Without escape codes, this yields $(k - 5)(|\Sigma| - 3)^{k-6}$ programs. When $k \rightarrow \infty$, $(|\Sigma| - 3)^k / ((k - 5)(|\Sigma| - 3)^{k-6}) = (|\Sigma| - 3)^6 / (k - 5) \rightarrow 0$.

That is, although string literals can represent information rather densely, they do not constitute the densest possible way of packing information into a C++

program (assuming the absence of escape codes). A pair of string literals yields an asymptotically strictly denser packing. Similarly, a triple of string literals is denser still, and so on. Counting the programs in the presence of escape codes is too difficult, but it seems likely that the phenomenon remains the same.

So string literals do not yield many enough programs. It seems difficult to first find a construct that does yield many enough programs, and then prove that it works.

4.2 End-of-file Data Segment Languages

In this section we prove a theorem that resembles Theorem 3, but relies on different assumptions and has a different proof.

We say that a three-way tester is *n-perfect* if and only if it does not answer “I don’t know” when the size of the instance is at most n . The following lemma is adapted from [11].

Lemma 1. *Each programming language has a constant e such that the size of each n -perfect three-way tester of variant E or S is at least $n - e$.*

Proof. Let T_n be any n -perfect three-way tester of variant E or S . Consider a program P that constructs character strings x in shortlex order and tests them with T_n until $T_n(x)$ replies “I don’t know”. If $T_n(x)$ replies “yes”, P simulates x before trying the next character string. When simulating x , P gives it the empty input in the case of variant E and x as the input in the case of S . The reply “I don’t know” eventually comes, because otherwise T_n would be a true halting tester. As a consequence, P eventually halts. Before halting, P simulates at least all halting programs of size at most n .

The time consumption of any simulated execution is at least the same as the time consumption of the corresponding genuine execution. So the execution of P cannot contain properly a simulated execution of P . P does not read any input, so it does not matter whether it is given itself or the empty string as its input. Therefore, the size of P is bigger than n . Because the only part of P that depends on n is T_n , there is a constant e such that the size of T_n is at least $n - e$. \square

In any everyday programming language, space characters can be added freely between tokens. Motivated by this, we define that a *blank character* is a character that, for any program, can be added to at least one place in the program without affecting the meaning of the program.

Theorem 4. *Let X be E or S . If the programming language is end-of-file data segment and has a blank character, then*

$$\forall T \in \text{three-way}(X) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{h}_T(n)}{p(n)} \geq c_T \wedge \frac{\bar{d}_T(n)}{p(n)} \geq c_T .$$

Proof. Assume first that tester T is a counter-example to the \bar{h}_T -claim. That is, for every $c > 0$, T has infinitely many values of n such that $\bar{h}_T(n)/p(n) < c$.

If T uses its data segment, let the use be replaced by the use of ordinary constants, liberating the data segment for the use described in the sequel. Let $T_{k,m}$ be the following program. Here k is a constant inside $T_{k,m}$ represented by $\Theta(\log k)$ characters, and m is the content of the data segment of $T_{k,m}$ interpreted as a natural number m in base $|\Sigma|$. Let a and d be the sizes of the actual program and data segment of $T_{k,m}$. We have $a = \Theta(\log k)$. Let x be the input of $T_{k,m}$.

The program $T_{k,m}$ first computes $n := k + d$. If $|x| < n$, then $T_{k,m}$ adds blank characters to x , to make its size n . Next, if $|x| > n$, then $T_{k,m}$ replies "I don't know" and halts. Otherwise $T_{k,m}$ gives x (which is now of size precisely n) to T . If $T(x)$ replies "yes" or "no", then $T_{k,m}$ gives the reply as its own reply and halts. Otherwise $T_{k,m}$ constructs each character string y of size n and tests it with T . $T_{k,m}$ simulates in parallel those y for which $T(y)$ returns "I don't know" until m of them have halted (with y or the empty string as the input, as appropriate). Then it aborts those that have not halted. If x is among those that halted, then $T_{k,m}$ replies "yes", otherwise $T_{k,m}$ replies "no".

For each $k \in \mathbb{N}$, there are infinitely many values of n such that $\bar{h}_T(n)/p(n) < |\Sigma|^{-k}$. For any such n we have $\bar{h}_T(n) < p(n)|\Sigma|^{-k} \leq |\Sigma|^n |\Sigma|^{-k}$. So $n - k$ characters suffice for representing $\bar{h}_T(n)$. Therefore, there is $T_{k,m}$ such that $d = n - k$ and $m = \bar{h}_T(n)$. It is an n -perfect three-way tester of size $a + d = d + \Theta(\log k) = n - k + \Theta(\log k)$. A big enough k yields a contradiction with Lemma 1.

The proof of the \bar{d}_T -claim is otherwise similar, but $T_{k,m}$ counts the number v of those y for which $T(y)$ returns "I don't know", and simulates the y until $v - m$ of them have halted. The \bar{h}_T -claim and \bar{d}_T -claim are combined into a single claim by choosing the smaller c_T and bigger n_T provided by their proofs. \square

4.3 End-of-file Dead Segment Languages

In this section we show that if dead information can be added extensively enough, a tester of variant E with an arbitrarily small positive failure rate exists, but the opposite holds for variant S. The reason for the result on variant E is that as the size of the programs grows, a bigger and bigger proportion of programs consists of copies of smaller programs. This phenomenon is so strong that to obtain the desired failure rate, it suffices to know the empty-input behaviour of all programs up to a sufficient size.

An *end-of-file dead segment language* is defined otherwise like end-of-file data segment language (Definition 1), but the actual program cannot read the data segment. This is the situation with any self-delimiting real-life programming language, whose compiler stops reading its input when it has read a complete program. Any end-of-file dead segment language is frequent and computationally domain-frequent.

Theorem 5. *For each end-of-file dead segment language,*

$$\forall c > 0 : \exists T_c \in \text{three-way}(E) : \forall n \in \mathbb{N} : \frac{\bar{h}_{T_c}(n) + \bar{d}_{T_c}(n)}{p(n)} \leq c.$$

The result also holds with approximating and generic-case testers.

Proof. Let $r(n)$ denote the number of programs whose dead segment is not empty. We have $r(n) \leq p(n) \leq |\Sigma|^n$, so $r(n)|\Sigma|^{-n} \leq 1$. For each $n \in \mathbb{N}$, $r(n+1) = |\Sigma|p(n) \geq |\Sigma|r(n)$. So $r(n)|\Sigma|^{-n}$ grows as n grows. These imply that there is ℓ such that $r(n)|\Sigma|^{-n} \rightarrow \ell$ from below when $n \rightarrow \infty$.

Because there are programs, $\ell > 0$. For every $c > 0$ we have $\ell c > 0$, so there is $n_c \in \mathbb{N}$ such that $r(n_c)|\Sigma|^{-n_c} \geq \ell - \ell c$. On the other hand, $p(n) = r(n+1)/|\Sigma| \leq \ell|\Sigma|^n$. These imply $p(n_c - 1)|\Sigma|^{n-n_c+1}/p(n) = r(n_c)|\Sigma|^{n-n_c}/p(n) \geq 1 - c$. Here $p(n_c - 1)|\Sigma|^{n-n_c+1}$ is the number of those programs of size n whose actual program is of size less than n_c .

The behaviour of a program on the empty input only depends on its actual program. Let n_a be the size of the actual program. Consider a three-way tester that looks the answer from a look-up table if $n_a < n_c$ and replies “I don’t know” if $n_a \geq n_c$ (cf. Proposition 3). It has $(\underline{h}_T(n) + \underline{d}_T(n))/p(n) \geq 1 - c$, implying the claim.

Proposition 1 generalizes the result to approximating and generic-case testers. \square

The above proof exploited the fact that the correct answer for a long program is the same as the correct answer for a similarly behaving short program. This does not work for testers of variant S, because the short and long program no longer get the same input, since each one gets itself as its input. Although the program does not have direct access to its dead segment, it gets it via the input. This changes the situation to the opposite of the previous theorem.

Theorem 6. *For each end-of-file dead segment language,*

$$\exists c > 0 : \forall T \in \text{three-way}(S) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n)}{p(n)} \geq c \wedge \frac{\bar{d}_T(n)}{p(n)} \geq c ,$$

$$\exists c > 0 : \forall T \in \text{generic}(S) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{d}_T(n)}{p(n)} \geq c , \text{ and}$$

$$\exists c > 0 : \forall T \in \text{approx}(S) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \geq c .$$

Proof. We prove first the claims on three-way and generic-case testers.

Let us recall the overall idea of the proof of Theorem 2. In that proof, for any tester T , a program P_T was constructed that gives its input x to T . If $T(x)$ replies “yes”, then $P_T(x)$ enters an eternal loop. If $T(x)$ replies “no”, then $P_T(x)$ halts immediately. To prove that a three-way tester has many hard (a) halting (b) non-halting instances, in the case of the “I don’t know” reply, $P_T(x)$ was made to (a) halt immediately (b) enter an eternal loop. All programs that halt on the same inputs as P_T were shown to be hard instances for T . For each n that is greater than a threshold that may depend on T , the existence of at least $c_T p(n)$ such programs was proven, where c_T may depend on T but not on n .

We now apply the same idea, but, to get a result where the same constant c applies to all testers T , we no longer construct a separate program P_T for each T .

Instead, we construct a single program P , which obtains T from the size of the input of P . (A similar idea appears in [4].) To discuss this, for any $i > 0$, let P_i be the program whose shortlex index is i . Let $\delta(i) = i - s(i) + 1$, where $s(i)$ is the biggest square number that is at most i . The essence of $\delta(i)$ is that as i gets the values $1, 2, 3, \dots$, $\delta(i)$ gets each value $1, 2, 3, \dots$ infinitely many times.

One more idea needs to be explained before discussing the details of P . Let Σ be partitioned to Σ_1 and Σ_2 of sizes $\lfloor \frac{|\Sigma|}{2} \rfloor$ and $\lceil \frac{|\Sigma|}{2} \rceil$. Let n_a be the size of the actual program of P . For each $n > n_a$, by modifying the dead segment, $|\Sigma|^{n-n_a}$ programs are obtained that have the same actual program as P . For $i \in \{1, 2\}$, let Π_i be the set of those of them whose dead segment ends with a character in Σ_i . We have $\frac{1}{3}|\Sigma|^{n-n_a} \leq |\Pi_1| \leq |\Pi_2|$. Because $0 < p(n) \leq |\Sigma|^n$, by choosing $c = \frac{1}{3}|\Sigma|^{n-n_a}$ we get $\frac{1}{3}|\Sigma|^{n-n_a}/p(n) \geq c$.

The program P first checks that its input x is a program with a non-empty dead segment. If it is not, then P halts immediately. Otherwise, P constructs $P_{\delta(|x|)}$ by going through all character strings in the shortlex order until $\delta(|x|)$ programs have been found. Then P constructs every program y that has the same size, has the same actual program, and belongs to the same Π_i as x . Then P executes the $P_{\delta(|x|)}(y)$ in parallel until any of the following happens.

If any $P_{\delta(|x|)}(y)$ replies “yes”, then P enters an eternal loop. If any $P_{\delta(|x|)}(y)$ replies “no”, then P aborts the remaining $P_{\delta(|x|)}(y)$ and halts. If every $P_{\delta(|x|)}(y)$ replies “I don’t know”, then P halts if $x \in \Pi_1$, and enters an eternal loop if $x \in \Pi_2$. If none of the above ever happens, then P fails to halt.

Recall that n_a is the size of the actual program of P . For any tester T , there are infinitely many n such that $n > n_a$ and $P_{\delta(n)}$ is T . For any such n , there are $|\Sigma|^{n-n_a}$ programs P' of size n that have the same actual program as P . Let P'' be any of them. The execution of $P(P'')$ starts $P_{\delta(n)}(P')$ for at least $\frac{1}{3}|\Sigma|^{n-n_a}$ distinct P' . If $P_{\delta(n)}(P')$ replies “yes”, then T claims that $P'(P')$ halts. Then also $P(P')$ halts, because P halts on the same inputs as P' , since they have the same actual program. Furthermore, $P(P'')$ halts, because P only looks at the size, actual program, and Π_i -class of its input, and P'' and P' agree on them. But the halting of $P(P'')$ is in contradiction with the behaviour of P described above. Therefore, no $P_{\delta(n)}(P')$ can reply “yes”. For a similar reason, none of them replies “no” either.

In conclusion, at least $\frac{1}{3}|\Sigma|^{n-n_a}$ distinct P' are hard instances for T . If T is a three-way tester, it replies “I don’t know” for all of them. Depending on whether $P'' \in \Pi_1$ or $P'' \in \Pi_2$, they are hard halting or hard non-halting instances. If T is a generic-case tester, it halts on none of these hard instances. Therefore, also $P(P'')$ and $P''(P'')$ fail to halt. So they all are hard non-halting instances.

In the case of approximating testers, P is modified such that it lets all $P_{\delta(|x|)}(y)$ run into completion and counts the “yes”- and “no”-replies that they give. If the majority of the replies are “no”, then P halts, otherwise P enters an eternal loop. For the same reasons as above, $P(P'')$ halts if and only if $P(P')$ halts if and only if $P'(P')$ halts. So at least half of the replies are wrong. \square

Finally, we prove a corollary of the above theorem that deals with *the halting problem itself*, not with imperfect testers. Imperfect testers are used in the proof

of the corollary, but not in the statement of the corollary.

Lemma 2. *Let X be any of E , S , and G , and let f be any total computable function from natural numbers to integers. If*

$$\exists c > 0 : \forall T \in \text{three-way}(X) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n)}{p(n)} \geq c ,$$

then $\lim_{n \rightarrow \infty} \frac{h(n) - f(n)}{p(n)}$ does not exist.

Proof. Assume that $\lim_{n \rightarrow \infty} (h(n) - f(n))/p(n) = x$ and $c > 0$. Let $i = \lceil -\log_2 c \rceil$. There is an x_i of the form $m + \sum_{j=1}^{i+1} b_j 2^{-j}$ such that m is an integer, $b_j \in \{0, 1\}$ when $1 \leq j \leq i+1$, and $x_i < x \leq x_i + 2^{-i-1}$. There also is n_0 such that when $n \geq n_0$, then $x_i \leq (h(n) - f(n))/p(n) < x_i + 2^{-i}$.

A tester T that disobeys the formula is obtained as follows. If $n < n_0$, T replies “I don’t know”. If $n \geq n_0$, T simulates all instances of size n until $\lceil x_i p(n) \rceil + f(n)$ have halted. If the given instance is among those that halted, then T replies “yes” and otherwise “I don’t know”. We have $\bar{h}_T(n)/p(n) < 2^{-i} \leq c$. \square

Corollary 2. *Consider variant S of the halting problem and any end-of-file dead segment language. Then $\lim_{n \rightarrow \infty} h(n)/p(n)$ does not exist.*

The proof of Lemma 2 can be modified to approximating testers with $(\bar{h}_T(n) + \bar{d}_T(n))/p(n) \geq c$. By (1), the limit fails to exist also in the framework of [4].

5 C++ without Comments and with Input

5.1 The Effect of Compile-Time Errors

We first show that among all character strings of size n , those that are not C++ programs — that is, those that yield a compile-time error — dominate overwhelmingly, as n grows. In other words, a random character string is not a C++ program except with vanishing probability. The result may seem obvious until one realizes that a C++ program may contain comments and string literals which may contain almost anything. We prove the result in a form that also applies to BF.

C++ is not self-delimiting. After a complete C++ program, there may be, for instance, definitions of new functions that are not used by the program. This is because a C++ program can be compiled in several units, and the compiler does not check whether the extra functions are needed by another compilation unit. Even so, if π is a C++ program, then $\pi 0$ is definitely not a C++ program and not even a prefix of a C++ program. Similarly, if π is a BF program, then $\pi]$ is not a prefix of a BF program.

Proposition 4. *If for every $\pi \in \Pi$ there is $c \in \Sigma$ such that $\pi c \notin \Gamma$, then*

$$\lim_{n \rightarrow \infty} \frac{p(n)}{|\Sigma|^n} = 0 .$$

Proof. Let $q(n) = |\Sigma^n \cap \Gamma|$. Obviously $0 \leq p(n) \leq q(n) \leq |\Sigma|^n$.

Assume first that for every $\varepsilon > 0$, there is $n_\varepsilon \in \mathbb{N}$ such that $p(n)/q(n) < \varepsilon$ for every $n \geq n_\varepsilon$. Because $0 \leq p(n)/|\Sigma|^n \leq p(n)/q(n)$, we get $p(n)/|\Sigma|^n \rightarrow 0$ as $n \rightarrow \infty$.

In the opposite case there is $\varepsilon > 0$ such that $p(n)/q(n) \geq \varepsilon$ for infinitely many values of n . Let them be $n_1 < n_2 < \dots$. Because πc is not a prefix of any program, $q(n_i + 1) \leq |\Sigma|q(n_i) - p(n_i) \leq (|\Sigma| - \varepsilon)q(n_i)$. For the remaining values of n , obviously $q(n + 1) \leq |\Sigma|q(n)$. These imply that when $n > n_i$, we have $0 \leq p(n)/|\Sigma|^n \leq q(n)/|\Sigma|^n \leq q(n_i)/|\Sigma|^{n_i} \leq (1 - \varepsilon/|\Sigma|)^i \rightarrow 0$ when $i \rightarrow \infty$, which happens when $n \rightarrow \infty$. \square

Consider a tester T that replies “no” if the compilation fails and “I don’t know” otherwise. If compile-time error is considered as non-halting, then Proposition 4 implies that $\underline{h}_T(n) \rightarrow 0$, $\bar{h}_T(n) \rightarrow 0$, $\underline{d}_T(n) \rightarrow 1$, and $\bar{d}_T(n) \rightarrow 0$ when $n \rightarrow \infty$. As we pointed out in Section 3.2, this is yet another instance of an anomaly stealing the result.

5.2 The C++ Language Model

The model of computation we study in this section is program–input pairs, where the programs are written in C++, and the inputs obey the rules stated by the Linux operating system. Furthermore, Σ is the set of all 8-bit bytes. To make firm claims about details, it is necessary to fix some language and operating system. The validity of the details below has been checked with C++ and Linux. Most likely many other programming languages and operating systems could have been used instead.

There are two deviations from the real everyday programming situation. First, of course, it must be assumed that unbounded memory is available. Otherwise everything would be decidable. (However, at any instant of time, only a finite number of bits are in use.) Second, it is assumed that the programs do not contain comments. This assumption needs a discussion.

Comments are information that is inside the program but ignored by the compiler. They have no effect to the behaviour of the compiled program. We show next that most long C++ programs consist of a shorter C++ program and one or more comments.

Lemma 3. *There are at most $(|\Sigma| - 1)^n$ comment-less C++ programs of size n .*

Proof. Everywhere inside a C++ program excluding comments, it is either the case that `@` or the case that the new line character `\n` cannot occur next. That is, for every character string α , either $\alpha@$ or $\alpha\n$ is not a prefix of any comment-less C++ program. \square

(Perhaps surprisingly, there indeed are places that are outside comments and where any byte except `\n` can occur.)

Lemma 4. *If $n \geq 16$, then there are at least $((|\Sigma| - 1)^4 + 1)^{(n-19)/4}$ C++ programs of size n .*

Proof. Let $A = \Sigma \setminus \{*\}$, and let $m = \lfloor n/4 - 4 \rfloor = \lceil (n - 19)/4 \rceil$. Consider the character strings of the form

`int main(){/* $\alpha\beta$ */}`

where α consists of $(n \bmod 4)$ space characters and β is any string of the form $\beta_1\beta_2\cdots\beta_m$, where $\beta_i \in A^4 \cup \{**/*\}$ for $1 \leq i \leq m$. Each such string is a syntactically correct C++ program of size n . Their number is $((|\Sigma| - 1)^4 + 1)^m \geq ((|\Sigma| - 1)^4 + 1)^{(n-19)/4}$. \square

Corollary 3. *The proportion of comment-less C++ programs among all C++ programs of size n approaches 0, when $n \rightarrow \infty$.*

Proof. Let $s = |\Sigma| - 1$. By Lemmas 3 and 4, the proportion is at most $s^n / (s^4 + 1)^{(n-19)/4} = s^{19} (s^4 / (s^4 + 1))^{(n-19)/4} \rightarrow 0$, when $n \rightarrow \infty$. \square

As a consequence, although comments are irrelevant for the behaviour of programs, they have a significant effect on the distribution of long C++ programs. To avoid the risk that they cause yet another anomaly stealing the result, we restrict ourselves to C++ programs without comments. This assumption does not restrict the expressive power of the programming language, but reduces the number of superficially different instances of the same program.

The input may be any finite string of bytes. This is how it is in Linux. Although not all such inputs can be given directly via the keyboard, they can be given by directing the so-called standard input to come from a file. There is a separate test construct in C++ for detecting the end of the input, so the end of the input need not be distinguished by the contents of the input. There are 256^n different inputs of size n .

The sizes of a program and input are the number of bytes in the program and the number of bytes in the input file. This is what Linux reports. The size of an instance is their sum. Analogously to Section 4.1, the size of a program is additional information to the concatenation of the program and the input. This is ignored by our notion of size. However, the notion is precisely what programmers mean with the word. Furthermore, the convention is similar to the convention in ordinary (as opposed to self-delimiting) Kolmogorov complexity theory [5].

Lemma 5. *With the C++ programming model in Section 5.2, $p(n) < |\Sigma|^{n+1}$.*

Proof. By Lemma 3, the number of different program–input pairs of size n is at most

$$\sum_{i=0}^n (|\Sigma| - 1)^i |\Sigma|^{n-i} = |\Sigma|^n \sum_{i=0}^n \left(\frac{|\Sigma| - 1}{|\Sigma|} \right)^i < |\Sigma|^n \sum_{i=0}^{\infty} \left(\frac{|\Sigma| - 1}{|\Sigma|} \right)^i = |\Sigma|^{n+1}.$$

\square

5.3 Proportions of Hard Instances

The next theorem says that with halting testers of variant G and comment-less C++, the proportions of hard halting and hard non-halting instances do not vanish.

Theorem 7. *With the C++ programming model in Section 5.2,*

$$\forall T \in \text{three-way}(G) : \exists c_T > 0 : \exists n_T \in \mathbb{N} : \forall n \geq n_T : \frac{\bar{h}_T(n)}{p(n)} \geq c_T \wedge \frac{\bar{d}_T(n)}{p(n)} \geq c_T .$$

Proof. We prove first the $\bar{h}_T(n)/p(n) \geq c_T$ part and then the $\bar{d}_T(n)/p(n) \geq c_T$ part. The results are combined by picking the bigger n_T and the smaller c_T .

There is a program P_T that behaves as follows. First, it gets its own size n_p from a constant in its program code. The constant uses some characters and thus affects the size of P_T . However, the size of a natural number constant m is $\Theta(\log m)$ and grows in steps of zero or one as m grows. Therefore, by starting with $m = 1$ and incrementing it by steps of one, it eventually catches the size of the program, although also the latter may grow.

Then P_T reads the input, counting the number of the characters that it gets with n_i and interpreting the string of characters as a natural number x in base $|\Sigma|$. We have $0 \leq x < |\Sigma|^{n_i}$, and any natural number in this range is possible. Let $n = n_p + n_i$.

Next P_T constructs every program–input pair of size n and tests it with T . In this way P_T gets the number $\underline{h}_T(n)$ of easy halting pairs of size n .

Then P_T constructs again every pair of size n . This time it simulates each of them in parallel until $\underline{h}_T(n) + x$ of them have halted. Then it aborts the rest and halts. It halts if and only if $\underline{h}_T(n) + x \leq h(n)$. (It may be helpful to think of x as a guess of the number of hard halting pairs.)

Among the pairs of size n is P_T itself with the string that represents x as the input. We denote it with (P_T, x) . The time consumption of any simulated execution is at least the same as the time consumption of the corresponding genuine execution. So the execution of (P_T, x) cannot contain properly a simulated execution of (P_T, x) . Therefore, either (P_T, x) does not halt, or the simulated execution of (P_T, x) is still continuing when (P_T, x) halts. In the former case, $h(n) < \underline{h}_T(n) + x$. In the latter case (P_T, x) is a halting pair but not counted in $\underline{h}_T(n) + x$, so $h(n) > \underline{h}_T(n) + x$. In both cases, $x \neq h(n) - \underline{h}_T(n)$.

As a consequence, no natural number less than $|\Sigma|^{n_i}$ is $\bar{h}_T(n)$. So $\bar{h}_T(n) \geq |\Sigma|^{n_i} = |\Sigma|^{n-n_p}$. By Lemma 5, $p(n) < |\Sigma|^{n+1}$. So for any $n \geq n_p$, we have $\bar{h}_T(n)/p(n) > |\Sigma|^{-n_p-1}$.

The proof of the $\bar{d}_T(n)/p(n) \geq c_T$ part is otherwise similar, except that P_T continues simulation until $p(n) - \underline{d}_T(n) - x$ pairs have halted. (Now x is a guess of $\bar{d}_T(n)$, yielding a guess of $h(n)$ by subtraction.) The program P_T gets $p(n)$ by counting the pairs of size n whose program part is compilable. It turns out that $p(n) - \underline{d}_T(n) - x \neq h(n)$, so x cannot be $\bar{d}_T(n)$, yielding $\bar{d}_T(n) \geq |\Sigma|^{n_i}$. \square

Next we adapt the second main result in [4] to our present setting, with a

somewhat simplified proof and obtaining the result also for three-way and generic-case testers.

Theorem 8. *With the C++ programming model in Section 5.2,*

$$\exists c > 0 : \forall T \in \text{three-way}(\mathbf{G}) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n)}{p(n)} \geq c \wedge \frac{\bar{d}_T(n)}{p(n)} \geq c ,$$

$$\exists c > 0 : \forall T \in \text{generic}(\mathbf{G}) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{d}_T(n)}{p(n)} \geq c , \text{ and}$$

$$\exists c > 0 : \forall T \in \text{approx}(\mathbf{G}) : \forall n_0 \in \mathbb{N} : \exists n \geq n_0 : \frac{\bar{h}_T(n) + \bar{d}_T(n)}{p(n)} \geq c .$$

Proof. The proof follows the same strategy as the proof of Theorem 6, but differs in some technical details.

To prove the claim for three-way testers, for any character string α , let $\text{lb}(\alpha) = 0$ if α is the empty string, and otherwise $\text{lb}(\alpha)$ is the value of the least significant bit of the last character of α . For any character strings α and β , let $\alpha \simeq \beta$ if and only if $|\alpha| = |\beta|$ and $\text{lb}(\alpha) = \text{lb}(\beta)$. For any size n greater than 0, “ \simeq ” has two equivalence classes, each containing $|\Sigma|^n/2$ character strings. For any $i > 0$, let P_i be the program whose shortlex index is i .

There is a program P that behaves as follows. We denote its execution on input α with $P(\alpha)$. Please observe that if $\alpha \simeq \beta$, then $P(\beta)$ behaves in the same way as $P(\alpha)$.

First $P(\alpha)$ finds the program $P_{\delta(|\alpha|)}$, where $\delta(i) = i - s(i) + 1$, where $s(i)$ is the biggest square number that is at most i .

Then $P(\alpha)$ goes through, in the shortlex order, all $\lceil |\Sigma|^{|\alpha|}/2 \rceil$ character strings β such that $\alpha \simeq \beta$, until any of the termination conditions mentioned below occurs or $P(\alpha)$ has gone through all of them. For each β , it runs $P_{\delta(|\alpha|)}$ on β . We denote this with $P_{\delta(|\alpha|)}(\beta)$. If $P_{\delta(|\alpha|)}(\beta)$ fails to halt, then $P(\alpha)$ never returns from it and thus fails to halt. If $P_{\delta(|\alpha|)}(\beta)$ halts replying “yes”, then $P(\alpha)$ enters an eternal loop, thus failing to halt. If $P_{\delta(|\alpha|)}(\beta)$ halts replying “no”, then $P(\alpha)$ halts immediately. If $P_{\delta(|\alpha|)}(\beta)$ halts replying “I don’t know”, then $P(\alpha)$ tries the next β . It is not important what $P(\alpha)$ does if $P_{\delta(|\alpha|)}(\beta)$ halts replying something else.

If $P_{\delta(|\alpha|)}(\beta)$ halted replying “I don’t know” for every β such that $\alpha \simeq \beta$, then $P(\alpha)$ checks whether $\text{lb}(\alpha) = 0$. If yes, then $P(\alpha)$ enters an eternal loop, otherwise $P(\alpha)$ halts.

Now let $T(Q, \gamma)$ be any three-way tester that tests whether program Q halts on the input γ . How the two components Q and γ of the input of T are encoded into one input string is not important. There is a program that has P hard-coded into a string constant, inputs β , calls $T(P, \beta)$, and gives its reply as its own reply. Let i be the shortlex index of this program, so the program is P_i .

There are infinitely many positive integers j such that $\delta(j) = i$. Let j be such, and let α be any character string of size j . So $P_{\delta(|\alpha|)}$ is P_i . If, during the execution of $P(\alpha)$, $P_i(\beta)$ ever replies “yes” or “no”, then the same happens during

the execution of $P(\beta)$, because $P(\beta)$ behaves in the same way as $P(\alpha)$ (the fact that $P_i(\beta)$ was called implies $\alpha \simeq \beta$). But that would be incorrect by the construction of P . Therefore, $T(P, \beta)$ replies “I don’t know” for every β of size j .

As a consequence, T has at least $|\Sigma|^j$ hard instances of size $|P| + j$. If $j > 0$, then half of them are halting and the other half non-halting, thanks to the $\text{lb}(\alpha) = 0$ test near the end of P . By Lemma 5, $p(n) < |\Sigma|^{n+1}$. So if $n = |P| + j > |P|$, then

$$\frac{\bar{h}_T(n)}{p(n)} \geq \frac{|\Sigma|^{n-|P|}}{2|\Sigma|^{n+1}} = \frac{1}{2|\Sigma|^{|P|+1}} \quad \text{and} \quad \frac{\bar{d}_T(n)}{p(n)} \geq \frac{1}{2|\Sigma|^{|P|+1}}.$$

The program P does not depend on n , so letting $c = 1/(2|\Sigma|^{|P|+1})$ we have the claim.

The proof for generic-case testers is otherwise similar, but the β are tried in parallel and $T(P, \beta)$ fails to halt for every β of size j . All hard instances are non-halting. The P for approximating testers lets each $P_{\delta(|\alpha|)}(\beta)$ continue until completion, counts the numbers of the “yes”- and “no”-replies they yield, and then does the opposite of the majority of the replies. \square

Application of Lemma 2 to this result yields the following.

Corollary 4. *With the C++ programming model in Section 5.2, $\lim_{n \rightarrow \infty} h(n)/p(n)$ does not exist.*

6 Conclusions

This study did not cover all combinations of a programming model, variant of the halting problem, and variant of the tester. So there is a lot of room for future work.

The results highlight what was already known since [6]: the programming model has a significant role. With some programming models, a phenomenon of secondary interest dominates the distribution of programs, making hard instances rare. Such phenomena include compile-time errors and falling off the left end of the tape of a Turing machine.

Many results were derived using the assumption that information can be packed very densely in the program or the input file. Sometimes it was not even necessary to assume that the program could use the information. It sufficed that the assumption allowed to make many enough similarly behaving longer copies of an original program. Intuition suggests that if the program can access the information, testing halting is harder than in the opposite case. A comparison of Theorem 5 to Theorem 6 supports this intuition.

Corollaries 2 and 4 and the comment after Corollary 2 tell that the proportion of *all* (not just hard) halting instances has no limit with end-of-file dead segment languages and variant S of the halting problem, with the C++ model and variant G, and in the framework of [4]. It must thus oscillate irregularly as the size of the program grows — irregularly because of Lemma 2. This is not a property of various notions of imperfect halting testers, but a property of the halting problem itself.

Acknowledgements

I thank professor Keijo Ruohonen for helpful discussions, and the anonymous reviewers of SPLST '13 and Acta Cybernetica for their helpful comments. The latter pointed out that Proposition 4 had been formulated incorrectly.

References

- [1] Calude, C. S. and Stay, M. A. Most programs stop quickly or never halt. *Advances in Applied Mathematics*, 40:295–308, 2008.
- [2] Hamkins, J. D. and Miasnikov, A. The halting problem is decidable on a set of asymptotic probability one. *Notre Dame Journal of Formal Logic*, 47(4):515–524, 2006.
- [3] Hopcroft, J. E. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [4] Köhler, S., Schindelhauer, C., and Ziegler, M. On approximating real-world halting problems. In Liśkiewicz, M. and Reischuk, R., editor, *Proc. 15th Fundamentals of Computation Theory*, Lecture Notes in Computer Science 3623, pages 454–466, 2005. Springer.
- [5] Li, M. and Vitányi, P. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2008.
- [6] Lynch, N. Approximations to the halting problem. *Journal of Computer and System Sciences*, 9:143–150, 1974.
- [7] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74:358–366, 1953.
- [8] Rybalov, A. On the strongly generic undecidability of the halting problem. *Theoretical Computer Science*, 377:268–270, 2007.
- [9] Schindelhauer, C. and Jakoby, A. The non-recursive power of erroneous computation. In Pandu Rangan, C., Raman, V., and Ramanujam, R., editors, *Proc. 19th Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 1738, pages 394–406, 1999. Springer.
- [10] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* ser. 2, 42:230–265, 1937.
- [11] Valmari, A. Sizes of up-to- n halting testers. In Halava, V., Karhumäki, J., and Matiyasevich, Y., editors, *Proceedings of the Second Russian Finnish Symposium on Discrete Mathematics*, TUCS Lecture Notes 17, pages 176–183, Turku, Finland, 2012.

- [12] Valmari, A. The asymptotic behaviour of the proportion of hard instances of the halting problem (extended version). Computer Science Research Repository arXiv:1307.7066, 2013.
- [13] Valmari, A. The asymptotic behaviour of the proportion of hard instances of the halting problem. In Kiss, Á., editor, *Proceedings of SPLST '13, 13th Symposium on Programming Languages and Software Tools*, pages 170–184, Szeged, Hungary, 2013.

Runtime Exception Detection in Java Programs Using Symbolic Execution*

István Kádár[†], Péter Hegedűs[†], and Rudolf Ferenc[†]

Abstract

Most of the runtime failures of a software system can be revealed during test execution only, which has a very high cost. In Java programs, runtime failures are manifested as unhandled runtime exceptions.

In this paper we present an approach and tool for detecting runtime exceptions in Java programs without having to execute tests on the software. We use the symbolic execution technique to implement the approach. By executing the methods of the program symbolically we can determine those execution branches that throw exceptions. Our algorithm is able to generate concrete test inputs also that cause the program to fail in runtime.

We used the Symbolic PathFinder extension of the Java PathFinder as the symbolic execution engine. Besides small example codes we evaluated our algorithm on three open source systems: jEdit, ArgoUML, and log4j. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system.

Keywords: Java runtime exception, symbolic execution, rule checking

1 Introduction

Nowadays, it is a big challenge of the software engineering to produce huge, reliable and robust software systems. About 40% of the total development costs go for testing [13], and the maintenance activities, particularly bug fixing of the system also require a considerable amount of resources [20]. Our purpose is to develop a new method and tool, which supports this phase of the software engineering lifecycle with detecting runtime exceptions in Java programs, and finding dangerous parts in the source code, that could behave as time-bombs during further development. The analysis will be done without executing the program in a real environment.

Runtime exceptions in the Java programming language are the instances of class `java.lang.RuntimeException`, which represent a sort of runtime error, for example

*This research was supported by the Hungarian national grant GOP-I.1.1-11-2011-0038 and the TÁMOP 4.2.4. A/2-11-1-2012-0001 European grant.

[†]University of Szeged, Department of Software Engineering Árpád tér 2. H-6720 Szeged, Hungary, E-mail: {ikadar|hpeter|ferenc}@inf.u-szeged.hu

an invalid type cast, an array over indexing, or division by zero. These exceptions are dangerous because they can cause a sudden stop of the program, as they do not have to be handled by the programmer explicitly.

Exploration of these exceptions is done by using a technique called symbolic execution [12]. When a program is executed symbolically, it is not executed on concrete input data but input data is handled as symbolic variables. When the execution reaches a branching condition containing a symbolic variable, the execution continues on both branches. This way, all of the possible branches of the program will be executed in theory. Java PathFinder (JPF) [10] is a software model checker which is developed at NASA Ames Research Center. In fact, Java PathFinder is a Java virtual machine that executes Java bytecode in a special way. Symbolic PathFinder (SPF) [14] is an extension of JPF, which can perform symbolic execution of Java bytecodes. The presented work is based on these tools.

The paper explains how the detection of runtime exceptions of the Java programming language was implemented using Java PathFinder and symbolic execution. Concrete input parameters of the method resulting a runtime exception are also determined. It is also described how the number of execution branches, and the state space have been reduced to achieve a better performance. The implemented tool called *Jpf Checker* has been tested on real life projects, the *log4j*, *ArgoUML*, and *jEdit* open source systems. We found multiple errors in the *log4j* system that were also reported as real bugs in its bug tracking system. The performance of the tool is acceptable since the analysis was finished in a couple of hours even for the biggest system used for testing.

The remainder of the paper is organized as follows. We give a brief introduction to symbolic execution in Section 2. After that in Section 3 we present our approach for detecting runtime exceptions. Section 4 discusses the results of the implemented algorithm on different small examples and real life open source projects. Section 5 collects the works that related to ours. Finally, we conclude the paper and present some future work in Section 6.

2 Symbolic Execution

During its execution, every program performs operations on the input data in a defined order. Symbolic execution [12] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When the execution reaches a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable, it cannot be evaluated, its value might be also true and false. The execution continues on both branches accordingly. This way we can simulate all the possible execution branches of the program.

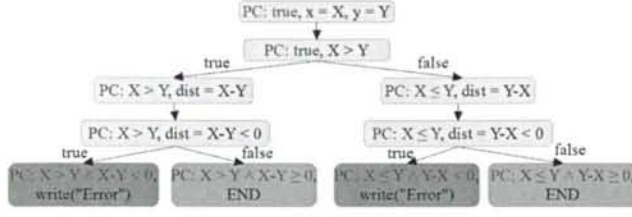
During symbolic execution we maintain a so-called *path condition (PC)*. The path condition is a quantifier-free logical formula with the initial value of true, and

```

1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");

```

(a)



(b)

Figure 1: (a) Sample code that determines the distance of two integers on the number line
(b) Symbolic execution tree of the sample code handling variable x and y symbolically

its variables are the symbolic variables of the program. If the execution reaches a branching condition that depends on one or more symbolic variables, the condition will be appended to the current PC with the logical operator *AND* to indicate the true branch, and the negation of the condition to indicate the false branch. With such an extension of the PC, each execution branch will be linked to a unique formula over the symbolic variables. In addition to maintaining the path condition, symbolic execution engines make use of the so called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula. Path condition can be solved at any point of the symbolic execution. Practically, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

All of the possible execution paths define a connected and acyclic directed graph called *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program. An example is shown in Figure 1.

Figure 1 (a) shows a sample code that determines the distance of two integers x and y . The symbolic execution of this code is illustrated on Figure 1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. The initial value of the path condition is true. Reaching the first if statement in line 3, there are two possibilities: the logical expression can be true or false; thus the execution branches and the logical expression and its negation is added to the PC as follows:

$$true \wedge X > Y \Rightarrow X > Y, \quad \text{and} \quad true \wedge \neg(X > Y) \Rightarrow X \leq Y$$

The value of variable $dist$ will be a symbolic expression, $X - Y$ on the true branch and $Y - X$ on the false one. As a result of the second if statement (line 8) the execution branches, and the appropriate PCs are appended again. On the true branches we get the following PCs:

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y$$

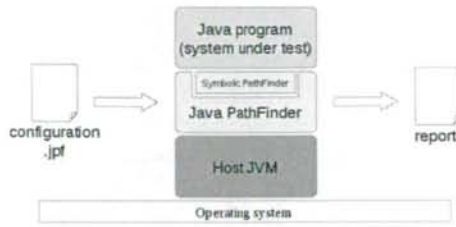


Figure 2: Java PathFinder as a virtual machine itself runs on a JVM, while performing a verification of a Java program

It is clear that these formulas are unsolvable, we cannot specify such X and Y that satisfy the conditions. This means that there are no such x and y inputs with which the program reaches the `write("Error")` statement. As long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned, there is no sense to continue the controversial execution.

It is impossible to explore all the symbolic states. It takes unreasonably long time to execute all the possible paths. A solution for this problem can be e.g. to limit the depth of the symbolic execution tree or the number of states which, of course, inhibit to examine all the states. The next subsection describes what are the available techniques in Symbolic PathFinder to address this problem.

2.1 Java PathFinder and Symbolic PathFinder

Java PathFinder (JPF) [10] is a highly customizable execution environment that aims at verifying Java programs. In fact, JPF is nothing more than a Java Virtual Machine which interprets the Java bytecode in a special way to be able to verify certain properties. It is difficult to determine what kind of errors can be found and which properties can be checked by JPF, it depends primarily on its configuration. The system has been designed from the beginning to be easily configurable and extendable. One of its extensions is *Symbolic PathFinder (SPF)* [14] that provides symbolic execution of Java programs by implementing a bytecode instruction set allowing to execute the Java bytecode according to the theory of symbolic execution.

JPF (and SPF) itself is implemented in Java, so it also have to run on a virtual machine, thus JPF is actually a middleware between the standard JVM and the bytecode. The architecture of the system is illustrated on Figure 2.

To start the analysis we have to make a configuration file with `.jpf` extension in which we specify different options as key-value pairs. The output is a report that contains e.g. the found defects. In addition to the ability of handling logical, integer and floating-point type variables as symbols, SPF can also handle complex types symbolically with the lazy initialization algorithm [11], and allows the symbolic execution of multi-threaded programs too.

SPF supports multiple constraint solvers and defines a general interface to communicate them. *Cvc3* is used to solve linear formulas, *choco* can handle non-linear

logical formulas too, while *IASolver* use interval arithmetic techniques to satisfy the path condition. Among the supported constraint solvers, *CORAL* proved to be the most effective in terms of the number of solved constraints and the performance [19].

To reduce the state space of the symbolic execution SPF offers a number of options. We can specify the maximum depth of the symbolic execution tree, and the number of elementary formulas in the path condition can also be limited. Further possibility is that with options *symbolic.minint*, *symbolic.maxint*, *symbolic.minreal*, and *symbolic.maxreal* we can restrict the value ranges of the integer and floating point types. With the proper use of these options the state space and the time required for the analysis can be reduced significantly.

3 Detection of Runtime Exceptions

We developed a tool that is able to automatically detect runtime exceptions in an arbitrary Java program. This section explains in detail how this analysis program, the JPF checker works.

To check the whole program we use symbolic execution, which is performed by Symbolic PathFinder. However, we do not execute the whole program symbolically to discover all of the possible paths, instead we symbolically execute the methods of the program one by one. Starting the analysis from the *main* method has several drawbacks. For example, the state space would be too large and we would need to cut it when the execution reaches the defined maximal depth in the symbolic execution tree. Our approach results in a significant reduction in the state space of the symbolic execution.

An important question is which variables to be handled symbolically. In general, execution of a method mainly depends on the actual values of its parameters and the referred external variables. Thus, these are the inputs of a method that should be handled symbolically to generally analyze it. Currently, we handle the parameters and data members of the class of the analyzed method symbolically.

Our goal is not only to indicate the runtime exceptions a method can throw (its type and the line causing the exception), but also to determine a parameterization that leads to throwing those exceptions. In addition, we determine this parameterization not only for the analyzed method which is at the bottom of the call stack, but for all the other elements in the call stack (i.e. recursively for all the called methods).

Our work can be divided into two steps:

1. It is necessary to create a runtime environment which is able to iterate through all the methods of a Java program, and start their symbolic execution using Symbolic PathFinder.
2. We need a JPF extension which is built on its listener mechanism, and which is able to indicate potential runtime exceptions and related parameterization while monitoring the execution.

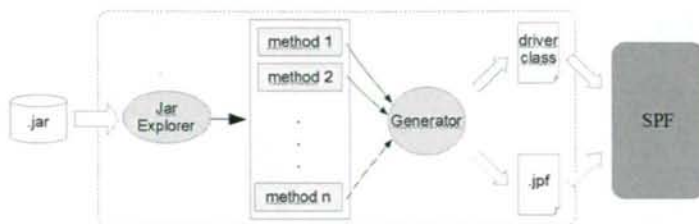


Figure 3: Architecture of the runtime environment

3.1 The Runtime Environment

The concept of the developers of Symbolic PathFinder was to start running the program in normal mode like in a real life environment, than at given points, e.g. at more complex or problematic parts in the program switch to symbolic execution mode [15]. The advantage of this approach is that, since the context is real, it is more likely to find real errors. E.g. the values of the global variables are all set, but if these variables are handled symbolically we can examine cases that never occur during a real run. A disadvantage is that it is hard to explore the problematic points of a program, it requires prior knowledge or preliminary work. Another disadvantage is that you have to run the program manually namely, that the control reach those methods which will be handled symbolic by the SPF.

In contrast, the tool we have developed is able to execute an arbitrary method or all methods of a program symbolically. The advantage of this approach is that the user does not have to perform any manual runs, the entire process can be automated. Additionally, the symbolic state space also remains limited since we do not execute the whole program symbolically, but their parts separately. The approach also makes it possible to analyze libraries that do not have a *main* method such as *log4j*. One of the major disadvantages is the that we back away from the real execution environment, which may lead to false positive error reports.

For implementing such an execution environment we have to achieve somehow that the control flow reaches the method we want to analyze. However, due to the nature of the virtual machine, JPF requires the entry point of the program, which is the class containing the main method. Therefore, we generate a driver class for each method containing a main method that only passes the control to the method we want to execute symbolically and carries out all the related tasks. Invoking the method is done using the Java Reflection API. We also have to generate a JPF configuration file that specifies, among others, the artificially created entry point and the method we want to handle symbolically. After creating the necessary files, we have to compile the generated Java class and finally, to launch Symbolic PathFinder.

The architecture of the system is illustrated in Figure 3. The input *jar* file is processed by the *JarExplorer*, which reads all the methods of the classes from the *jar* file and creates a list from them. The elements of the list is taken by the *Generator* one by one. It generates a driver class and a JPF configuration file for


```

1. exceptionThrown() {
2.   exception = getPendingException();
3.   if (isInstanceOfRuntimeException(exception)) {
4.     pc = getCurrentPc();
5.     solve(pc);
6.     summary = new FoundExceptionSummary();
7.     summary.setExceptionType(exception);
8.     summary.setThrownFrom(exception);
9.     summary.setParameterization(parsePc(pc, analyzedMethod));
10.    invocationChain = buildInvocationChain();
11.    foreach(Method m : invocationChain) {
12.      summary.addStackTraceElement(m, parsePc(pc, m));
13.    }
14.    foundExceptions.add(summary);
15.  }
16.}

```

Figure 4: Pseudo code of the exceptionThrown event

each method. After the generation is complete, we start the symbolic execution.

3.2 Implementing a Listener Class

During functioning, JPF sends notifications about certain events. This is realized with so-called listeners, which are based on the observer design pattern. The registered listener objects are notified about and can react to these events. JPF can send notifications of almost every detail of the program execution. There are low-level events such as execution of a bytecode instruction, as well as high-level events such as starting or finishing the search in the state space. In JPF, basically two listener interfaces exist: the *SearchListener* and *VMLListener* interface. While the former includes the events related to the state space search, the latter reports the events of the virtual machine. Because these interfaces are quite large and the specific listener classes often implement both of them, adapter classes are introduced that implement these interfaces with empty method bodies. Therefore, to create our custom listener we derived a class from this adapter and implemented the necessary methods only.

Our algorithm for detecting runtime exceptions is briefly summarized below. By performing symbolic execution of a method all of its paths are executed, including those that throw exceptions. When an exception occurs, namely when the virtual machine executes an *ATHROW* bytecode instruction, JPF triggers and *exceptionThrown* event. Thus, we implemented the *exceptionThrown* method in our listener class. Its pseudo code is shown in Figure 4.

First, we acquire the thrown Exception object (line 2), then we decide whether it is a runtime exception (i.e. whether it is an instance of the class *RuntimeException*) (line 3). If it is, we request the path condition related to the actual path and

use the constraint solver to find a satisfactory solution (lines 4-5). Lines 6-9 set up a summary report that contains the type of the thrown exception, the line that throws it and a parameterization which causes this exception to be thrown. The parameterization is constructed by the *parsePC()* method, which assigns the satisfactory solutions of the path condition to the method parameters. Lines 10-13 take care of collecting and determining parameterization for the methods in the call stack. If the source code does not specify any constraint for a parameter on the path throwing an exception (i.e. the path condition does not contain the variable), then there is no related solution. This means that it does not matter what the actual value of that parameter is, as it does not affect the execution path, and the method is going to throw an exception due to the values of other parameters. In such cases *parsePc()* method assigns the value “any” to these parameters.

It is also possible that a parameter has a concrete value. Figure 5 illustrates such an example. When we start the symbolic execution of method *x()*, its parameter *a* is handled symbolically. As *x()* calls *y()* its parameter *a* is still a symbol, but *b* is a concrete value (42). In a case like this, *parsePc()* have to get the concrete value from the stack of the actual method.

```
1. void x(int a) {
2.     short b = 42;
3.     y(a, b);
4. }
5. void y(int a, short b) {
6.     ...
7.     throw new NullPointerException();
8.     ...
9. }
```

Figure 5: An example call with both symbolic and concrete parameters

We note that the presented algorithm reports any runtime exceptions regardless of the fact whether it is caught by the program or not. The reason of this is that we think that relying on runtime exceptions is a bad coding practice and a runtime exception can be dangerous even if it is handled by the program. Nonetheless, it would be easy to modify our algorithm to detect uncaught exceptions only.

4 Results

The developed tool was tested in a variety of ways. The section describes the results of these test runs. We analyzed manually prepared example codes containing instructions that cause runtime exceptions on purpose; then we performed analysis on different open-source software to show that our tool is able to detect runtime exceptions in real programs, not just in artificially made small examples. The subject systems are the log4j (<http://logging.apache.org/log4j/>) logging library, the ArgoUML modeling tool (<http://argouml.tigris.org/>), and the jEdit text editor program (<http://www.jedit.org/>). We prove the validity of the detected exceptions by the bug reports, found in the bug tracking systems of these projects, that describe program faults caused by those runtime exceptions that are also found by the developed tool.

```

    public class Example5 {
    ...
8. void callRun(int x, int y) {
9.     Integer i = null;
10.    if (x > 6) {
11.        int b = 9;
12.        run(b, y);
13.        i = Integer.valueOf(b);
14.        System.out.println(i);
15.    } else {
16.        i = Integer.valueOf(3);
17.        System.out.println(i);
18.    }
19. }

20. public void run(int x, int y) {
21.    if (y > 10) {
22.        int[] arr = new int[5];
23.        for (int i = 0; i < x; i++) {
24.            arr[i] = i;
25.        }
26.    } else {
27.        Integer i = null;
28.        if (y < 5) {
29.            i = Integer.valueOf(4);
30.            i.floatValue();
31.        } else {
32.            System.out.println(
33.                i.floatValue());
34.        }
35.    }
36. }}

```

Figure 6: Manually prepared example code with the analysis of method `callRun()`

4.1 Manually Prepared Examples

A small manually prepared example code is shown on Figure 6. The method under test is `callRun()` which calls method `run()` in line 12. Running our algorithm on this code gives two hits: the first is an `ArrayIndexOutOfBoundsException`, the second is a `NullPointerException`. The first exception is thrown by method `run()` at line 24. A parametrization leading to this exception is `callRun(7, 11)`. Method `run()` will be called only if $x > 6$ (line 10) that is satisfied by 7 and it is called with the concrete value 9 and symbol y . At this point there is no condition for y . Method `run()` can reach line 24 only if $y > 10$, the indicated value 11 is obtained by satisfying this constraint. Throwing of the `ArrayIndexOutOfBoundsException` is due to the fact that in line 22 we declare a 5-element array but the following for loop runs from 0 to x . The value of x at this point is 9 which leads to an exception.

The train of thought is similar in the case of the second exception. The problem is that variable i created in line 27 initialized only in line 29 to a value different from `null`, but not in the else block, therefore line 33 throws a `NullPointerException`. This requires that the value of y not to be greater than 10 and not to be less than 5. These restrictions are satisfied by e.g. 5, and value 7 for x is necessary to invoke `run()`. So the parametrizations are `callRun(7, 5)` and `run(9, 5)`. The analysis is finished in less than a second.

A second example code is presented in Figure 7. The resulting report refers to an `ArithmeticException`, which is thrown at line 39 and the stack trace highlights that the problematic method is `expand()` which is invoked at line 30 by `run()`. The control flow reaches line 30 only if variable b is false. For example, if n is -999, and `check` has the value true, as the parameter list in the error report included, b will be false and the `expand()` method on the else branch will be executed. At


```

...
3. public class Example3 {
    ...
8.     public void run(int n,
        boolean check, A a) {
9.         boolean b = check && n >= 0;
10.        int max = Integer.MIN_VALUE;
11.        if (b) {
12.            if (a != null) {
13.                int l = n;
14.                int r = 2*n + 1;
15.                if (a.getMember() > 120) {
16.                    if (l <= a.getMember()) {
17.                        max = a.getMember();
18.                    } else {
19.                        max = l;
20.                    }
21.                    if (r > max) {
22.                        max = r;
23.                    }
24.                    while (max < n) {
25.                        max = expand(n, 0);
26.                    }
27.                }
28.            }
29.        } else {
30.            max = expand(n, 0);
31.        }
32.        System.out.println("Maximum"
33.            + value: " + max);
34.    }

35. private int expand(int n, int m) {
36.     double res = count(m);
37.     if (res > n) {
38.         do {
39.             res = n / res;
40.             res -= 2;
41.         } while (res >= 0);
42.         return n + m;
43.     } else {
44.         return (int)res;
45.     }
46. }
47.
48. private int count(int l) {
49.     int count = 1;
50.     for (int i=100; i>0; i--) {
51.         if (i % 3 == 0) {
52.             count++;
53.         }
54.     }
55.     return count;
56. }
57.
58. }

1. public class A extends Letter {
2.     ...
3.     public int member;
4.
5.     public int getMember() {
6.         return member;
7.     }
8.     ...
9. }

```

Figure 7: Manually prepared example code with the analysis of method `run()`

line 36, variable `res` has a concrete value because method `count()` will be executed. It can be seen that `res` is definitely a non-negative integer, thus the condition at line 37 is true if $n = -999$. Then the loop begins to execute, and variable `res` will be reduced to 0 after a number of iterations, leading to a division by 0 fault. In the report, the third parameter of the examined `run()` method is “any”. That is because this parameter does not play a role in whether or not the program runs onto the discussed `ArithmeticException`.

Line 25 in method `run()` also calls `expand()`, but there is no corresponding error report. In fact, due to the instructions at lines 13-23, the condition at line 24 is always false, thus this `expand()` call will never be executed. Actually, line 25 is unreachable code.

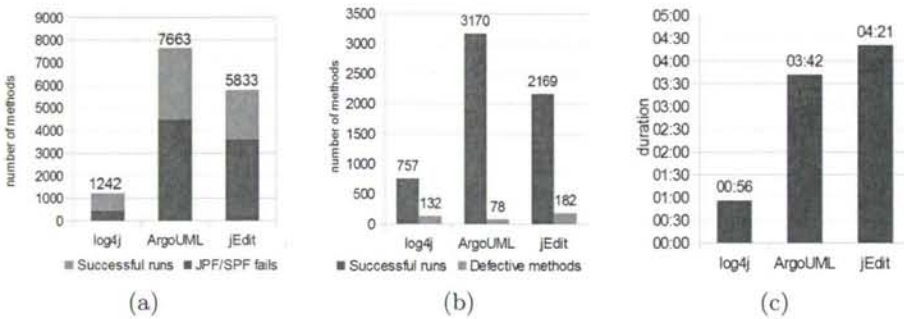


Figure 8: (a) Number of methods examined in the programs and the number of JPF or SPF faults (b) Number of successfully analyzed methods and the number of defective methods (c) Analysis time

4.2 Analysis of Open-source Systems

Analysis of log4j 1.2.15, ArgoUML 0.28 and jEdit 4.4.2 were carried out on a desktop computer with an Intel Core i5-540M 2.53 GHz processor and 8 GB of memory. In all three cases the analysis was done by executing all the methods of the release jar files of the projects symbolically.

Figure 8 (a) displays the number of methods we analyzed in the different programs. We started analyzing 1242 methods in log4j of which only 757 were successful, in 474 cases the analysis stopped due to the failure of the Java PathFinder (or Symbolic PathFinder). There are a lot of methods in ArgoUML which also could not be analyzed, more than half of the checks ended with failure. In case of jEdit the ratio is very similar. Unfortunately, in general JPF stopped with a variety of error messages.

Despite the frequent failures of JPF, our tool indicated a fairly large number of runtime exceptions in all three programs. Figure 8 (b) shows the number of successfully analyzed methods and the methods with one or more runtime exceptions. The hit rate is the highest for log4j and despite its high number of methods, relatively few exceptions were found in ArgoUML.

The analysis times are shown in Figure 8 (c). Analysis of log4j completed within an hour, while analysis of ArgoUML, that contains more than 7500 methods, took 3 hours and 42 minutes. Although jEdit contains fewer methods than ArgoUML, its full analysis were more time-consuming. The performance of our algorithm is acceptable, especially considering that the analysis was performed on an ordinary desktop PC not on a high-performance server. However, it can be assumed that the analysis time would grow with less failed method analysis.

It is important to note, that not all indicated exceptions are real errors. This is because the analysis were performed in an artificial execution environment which might have introduced false positive hits. When we start the symbolic execution of a method we have no information about the circumstances of the real invocation. All parameters and data members are handled symbolically, that is, it is considered

```

    public class SimpleLayout extends Layout {
        ...
58.     public String format(LoggingEvent event) {
59.
60.         sbuf.setLength(0);
61.         sbuf.append(event.getLevel().toString());
62.         sbuf.append(" - ");
63.         sbuf.append(event.getRenderedMessage());
64.         sbuf.append(LINE_SEP);
65.         return sbuf.toString();
66.     }
        ...
    }
    public class LoggingEvent implements java.io.Serializable {
        ...
        transient public Priority level;
        ...
255.     public Level getLevel() {
256.         return (Level) level;
257.     }
    }
    public class Level extends Priority implements Serializable{
        ...
    }

```

Figure 9: Method `org.apache.log4j.SimpleLayout.format()` and its environment.

that their value can be anything although it is possible that a particular value of a variable never occurs.

Despite the fact that not all the reported exceptions are real program errors they are definitely representing real risks. During the modification of the source code there are inevitably changes that introduce new errors. These errors often appear in form of runtime exceptions (i.e. in places where our algorithm found possible failures). So the majority of the reported exceptions do not report real errors, but potential sources of danger that should be paid special attention.

In the following, we are going to show some interesting faults found by our tool in the above systems.

The first example method is `org.apache.log4j.SimpleLayout.format()` of `log4j`, which is shown in Figure 9. In this method three possible runtime exceptions are found by the tool. The first two are `NullPointerExceptions`, both thrown at line 61. The produced report says that the first NPE will be thrown if the parameter is *null*, and the second when this parameter differs from *null*. In the first case, when the parameter is *null*, expression `event.getLevel()` causes the exception, since a method of a *null* reference cannot be called. When parameter *event* is not *null*, the code gets the *level* data member and calls its `toString()` method. The second `NullPointerException` is caused by the fact that the requested *level* data member


```

public class FindDialog extends ArgDialog ... { ... }
class PredicateMType extends PredicateType {
    ...
727. public static PredicateType create(Object c0, Object c1, Object c2) {
728.     Class[] classes = new Class[3];
729.     classes[0] = (Class) c0;
730.     classes[1] = (Class) c1;
731.     classes[2] = (Class) c2;
732.     return new PredicateMType(classes);
733. }
    ...
}

```

Figure 10: Method `org.argouml.ui.PredicateMType.create()`

can also be null, thus using operator ‘.’ may raise the exception.

The third exception is a `ClassCastException`. As shown, at line 256 in class *LoggingEvent* there is a type cast which tries to convert the *level* member which has a type *Priority* to a *Level* object. According to the listing in the bottom of Figure 9, class *Level* is a descendant of class *Priority*, thus the cast at line 256 is a downcast, which is incorrect in case the dynamic type of the member is not *Level*.

Three possible `ClassCastException`s are revealed in method *PredicateMType.create()* that is depicted in Figure 10. Lines 729, 730 and 731 cast down the three parameters from *Object* to *Class* without performing any type check. The first entry in the report says that *create(null, null, !null)* parametrization can lead to an exception thrown at line 731. If *c0* and *c1* parameters are null, lines 729 and 730 are executed without any problem, because casting a null reference to any class is permitted in Java. It is important that this does not mean that *c0* and *c1* have to be necessarily null, the report just gives a sample parametrization which leads the execution to the exception. As long as the third parameter is not null a `ClassCastException` can be raised. Of course, to achieve this it is necessary that the parameter type is different from *Class*. Parametrization *create(null, !null, “any”)* leads to potential fault at line 730. The reasoning is similar to the previous one: if *c0* is null and *c1* is non-null (and of course it is not a *Class*) `ClassCastException` will be thrown. The third parameter is completely irrelevant. In case of the third `ClassCastException`, occurring at line 729, the values of *c1* and *c2* do not matter.

The last example is a tiny method, *MRUFileManager.getFile()* shown in Figure 11. At line 98, *getFile()* checks whether the *index* parameter is less then the size of the *_mruFileList* *LinkedList*. If so, the return value is the corresponding element of the *LinkedList*, otherwise *null*. Our report shows that the *index* can be a negative number, too. This case is not handled, and *LinkedList.get()* will throw an *IndexOutOfBoundsException* if method *getField()* is called for example with -999. Calling *getField()* with a negative number seems unreasonable and of course it is, but possible.

```

public class MRUFileManager {
    ...
    private LinkedList _mruFileList;
    ...
    public int size() {
        return _mruFileList.size();
    }
97.  public Object getFile(int index) {
98.      if (index < size()) {
99.          return _mruFileList.get(index);
100.      }
101.
102.      return null;
103.  }
    ...
}

```

Figure 11: Method `org.apache.log4j.lf5.viewer.configure.MRUFileManager.getFile()`

4.3 Real Errors

In this subsection a few defects are presented which are reported in bug tracking systems, and caused by runtime exceptions found also by our tool. The first affected bug¹ reports the termination of an application using log4j version 1.2.14 caused by a `NullPointerException`. The reporter got the Exception from line 59 of *ThrowableInformation.java* thrown by method *org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()* as shown in the given stack trace. The code of the method and the problematic line detected by our analysis is shown in Figure 12.

The problem here is that the initialization of the *throwable* data member of class *ThrowableInformation* is omitted, its value is null causing a `NullPointerException` at line 59. This causes that the *log()* method of log4j can also throw an exception which should never happen. Our tool found other errors as well which demonstrate its strength of being capable of detecting real bugs.

The next exception is also a `NullPointerException`, which occurred in log4j 1.2.15. The bug report² explains that the runtime exception causing the halt comes from method *org.apache.log4j.NDC.remove()*, at line 377. Figure 13 shows the corresponding piece of code. The fault here is that the *ht* static data member is null. Although the data member is initialized as Figure 13 shows, it is possible that during the execution its value is set to null. The report in the log4j bug tracking system sheds light to this. The reporter also mentions that according to his observations, the other methods of class *NDC*, which use the *ht* member, first check whether it is null or not, but in method *remove()* there is no such investigation.

¹https://issues.apache.org/bugzilla/show_bug.cgi?id=44038

²https://issues.apache.org/bugzilla/show_bug.cgi?id=45335

```

public class ThrowableInformation implements java.io.Serializable {
    private transient Throwable throwable;
    ...
54. public String[] getThrowableStrRep() {
55.     if(rep != null) {
56.         return (String[]) rep.clone();
57.     } else {
58.         VectorWriter vw = new VectorWriter();
59.         throwable.printStackTrace(vw);
60.         rep = vw.toStringArray();
61.         return rep;
62.     }
63. }
    ...
}

```

Figure 12: Method `org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()`

We describe one more error that was also found in log4j version 1.2.15³. The error is at line 312 of the class `org.apache.log4j.net.SyslogAppender`. The line is inside the method `append()` in which there is a `NullPointerException` again. The code snippet is shown in Figure 14.

The reason of this runtime error is that the *layout* data member, which is inherited from class `AppenderSkeleton`, stays uninitialized. Our report also includes a `ClassCastException` thrown by method `getLevel()` at line 294. This fault is the same that we already described explaining Figure 9 in the previous subsection.

5 Related Work

In this section we present works that are related to our research. First, we introduce some well-known symbolic execution engines, then we show the possible applications of the symbolic execution. We also summarize the problems that have been solved successfully by Symbolic PathFinder that we used for implementing our approach. Finally, we present the existing approaches and techniques for runtime exception detection.

The idea of symbolic execution is not new, the first publications and execution engines appeared in the 1970's. One of the earliest work is by King that lays down the fundamentals of symbolic execution [12] and presents the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another work from the 1970's by Boyer et al. presents a similar system called SELECT [1] that can be used for executing LISP programs

³https://issues.apache.org/bugzilla/show_bug.cgi?id=46271


```

public class NDC {
    ...
    static Hashtable ht = new Hashtable();
    ...
374. static
375. public
376. void remove() {
377.     ht.remove(Thread.currentThread());
378.
379.     // Lazily remove dead-thread references in ht.
380.     lazyRemove();
381. }
    ...
}

```

Figure 13: Source code of method `org.apache.log4j.NDC.remove()`

symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation; in addition, for every path it gives back the path condition over the symbolic variables.

Starting from the last decade the interest about the technique is constantly growing, numerous programs have been developed that aim at dynamic test input generation using symbolic execution. The EXE (EXecution generated Executions) [3] presented by Cadar et al. at the Stanford University is an error checking tool made for generating input data on which the program terminates with failure. The input generation is done by the STP built-in constraint solver that solves the path condition of the path causing the failure. EXE achieved promising results on real life systems. It found errors in the package filter implementations of *BSD* and *Linux*, in the *udhcpd* DHCP server and in different Linux file systems. The runtime detection algorithm presented in this work solves the path condition to generate test input data similarly to EXE. The basic difference is that for running EXE one needs to declare the variables to be handled symbolically while for Jpf Checker there is no need for editing the source code before detection.

The DART [7] (Directed Automata Random Testing) by Godefroid et al. tries to eliminate the shortcomings of the symbolic execution e.g. when it is unable to handle a condition due to its unlinear nature. DART executes the program with random or predefined input data and records the constraints defined by the conditions on the input variables when it reaches a conditional statement. In the next iteration taking into account the recorded constraints it runs the program with input data that causes a different execution branch of the program. The goal is to execute all the reachable branches of the program by generating appropriate input data. The CUTE and jCUTE systems [16] by Sen and Agha extend DART with multithreading and dynamic data structures. The advantage of these tools is that they are capable of handling complex mathematical conditions due to concrete

```

public abstract class AppenderSkeleton {
    protected Layout layout;
    ...
}
public class SyslogAppender extends AppenderSkeleton {
    SyslogQuietWriter sqw;
    private boolean layoutHeaderChecked = false;
    ...
291. public
292. void append(LoggingEvent event) {
293.
294.     if(!isAsSevereAsThreshold(event.getLevel()))
295.         return;
296.
297.     // We must not attempt to append if sqw is null.
298.     if(sqw == null) {
299.         errorHandler.error("No syslog host is set for SyslogAppender"
300.                             + named " + this.name + ".");
301.         return;
302.     }
303.
304.     if (!layoutHeaderChecked) {
305.         if (layout != null && layout.getHeader() != null) {
306.             sendLayoutMessage(layout.getHeader());
307.         }
308.         layoutHeaderChecked = true;
309.     }
310.
311.
312.     String packet = layout.format(event);
313.     String hdr = getPacketHeader(event.timeStamp);
314.
315.     if(facilityPrinting || hdr.length() > 0) {
316.         StringBuffer buf = new StringBuffer(hdr);
317.         if(facilityPrinting) {
318.             buf.append(facilityStr);
319.         }
320.         buf.append(packet);
321.         packet = buf.toString();
322.     }
    ...
}}

```

Figure 14: Source code of method `org.apache.log4j.net.SyslogAppender.append()`

executions. This can be also achieved in Jpf Checker by using the concolic execution of SPF; however, symbolic execution allows a more thorough examination of the source code. Further description and comparison of the above mentioned tools can be found e.g. in the work of Coward [4].

There are also approaches and tools for generating test suites for .NET programs using symbolic execution. Pex [21] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [9], which aims to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

Song et al. applied the symbolic execution to the verification of networking protocol implementations [18]. The SymNV tool creates network packages with which a high coverage can be achieved in the source code of the daemon, therefore potential rule violations can be revealed according to the protocol specifications.

The SAFELI tool [6] by Fu and Qian is a SQL injection detection program for analyzing Java web applications. It first instruments the Java bytecode then executes the instrumented code symbolically. When the execution reaches a SQL query the tool prepares a string equation based on the initial content of the web input components and the built-in SQL injection attack patterns. If the equation can be solved the calculated values are used as inputs which the tool verifies by sending a HTML form to the server. According to the response of the server it can decide whether the found input can be a real attack or not.

The main application of the Java PathFinder and its symbolic execution extension is the verification of the internal projects in NASA. Bushnell et al. describes the application of Symbolic PathFinder in TSAFE (Tactical Separation Assisted Flight Environment) [2] that verifies the software components of an air control and collision detection system. The primary target is to generate useful test cases for TSAFE that simulates different wind conditions, radar images, flight schedules, etc.

The detection of design patterns can be performed using dynamic approaches as well as with static program analysis. With the help of a monitoring software the program can be analyzed during manual execution and conclusions about the existence of different patterns can be made based on the execution branches. In his work, von Detten [22] applied symbolic execution with Symbolic PathFinder supplementing manual execution. This way, more execution branches can be examined and the instances found by traditional approaches can be refined.

Ihantola [8] describes an interesting application of JPF in education. He generates test inputs for checking the programs of his students. His approach is that functional test cases based on the specification of the program and their outcome (successful or not) is not enough for educational purposes. He generates test cases for the programs using symbolic execution. This way the students can get feedbacks like "the program works incorrectly if variable a is larger than variable b plus 10".

Sinha et al. deal with localizing Java runtime errors [17]. The introduced approach aims at helping to fix existing errors. They extract the statement that threw the exception from its stack trace and perform a backward dataflow analysis starting from there to localize those statements that might be the root causes of the exception.

The work of Weimer and Necula [23] focuses on proving safe exception handling in safety critical systems. They generate test cases that lead to an exception by violating one of the rules of the language. Unlike Jpf Checker they do not generate test inputs based on symbolic execution but solving a global optimization problem on the control flow graph (CFG) of the program.

The JCrasher tool [5] by Csallner and Smaragdakis takes a set of Java classes as input. After checking the class types it creates a Java program which instantiates the given classes and calls each of their public methods with random parameters. This algorithm might detect failures that cause the termination of the system such as runtime exceptions. The tool is capable of generating JUnit test cases and can be integrated to the Eclipse IDE. Similarly to Jpf Checker JCrasher also creates a driver environment but it can analyze public methods only and instead of symbolic execution it generates random data which is obviously not feasible for examining all possible execution branches.

6 Conclusions and Future Work

The introduced approach for detecting runtime exceptions works well not just on small, manually prepared examples but it is able to find runtime exceptions which are the causes of some documented runtime failures (i.e. there exists an issue for them in the bug tracking system) in real world systems also. However, not all the detected possible runtime exceptions will actually cause a system failure. There might be a large number of exceptions that will never occur running the system in real environment. Nonetheless, the importance of these warnings should not be underrated since they draw attention to those code parts that might turn to real problems after changing the system. Considering these possible problems could help system maintenance and contributes to achieving a better quality software. As we presented in Section 4 the analysis time of real world systems are also acceptable, therefore our approach and tool can be applied in practice.

Unfortunately the Java PathFinder and its Symbolic PathFinder extension – which we used for implementing our approach – contain a lot of bugs. It made the development very troublesome, but the authors at the NASA were really helpful. We contacted them several times and got responses very quickly; they fixed some blocker issues particularly for our request. Although JPF and SPF have several bugs, it is under constant development and becoming more and more stable.

The achieved results are very promising and we continue the development of our tool. Our future plan is to eliminate the false positive and those hits that are irrelevant. We would also like to provide more details about the environment of the method in which the runtime exception is detected. The implemented tool gives

only the basic information about the reference type parameters whether they are *null* or not, and we cannot tell anything about the values of the member variables of the class playing a role in a runtime exception. These improvements of the algorithm are also in our future plans.

The presented approach is not limited to runtime exception detection. We plan to utilize the potentials of the symbolic execution by implementing other types of error and rule violation checkers. E.g. we can detect some special types of infinite loops, dead or unused code parts, or even SQL injection vulnerabilities.

References

- [1] Boyer, Robert S., Elspas, Bernard, and Levitt, Karl N. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.
- [2] Bushnell, D., Giannakopoulou, D., Mehltitz, P., Paielli, R., and Păsăreanu, Corina S. Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. In *Aerospace Conference, 2009 IEEE*, pages 1–10, 2009.
- [3] Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., and Engler, Dawson R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06*, pages 322–335, New York, NY, USA, 2006. ACM.
- [4] Coward, P. David. Symbolic Execution Systems – a Review. *Software Engineering Journal*, 3(6):229–239, November 1988.
- [5] Csallner, Christoph and Smaragdakis, Yannis. JCrasher: an Automatic Robustness Tester for Java. *Software Practice and Experience*, 34(11):1025–1050, September 2004.
- [6] Fu, Xiang and Qian, Kai. SAFELI: SQL Injection Scanner Using Symbolic Execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, TAV-WEB '08*, pages 34–39, New York, 2008. ACM.
- [7] Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, pages 213–223, New York, NY, USA, 2005. ACM.
- [8] Ihantola, Petri. Test Data Generation for Programming Exercises with Symbolic Execution in Java Pathfinder. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Baltic Sea '06*, pages 87–94, New York, 2006. ACM.

- [9] Jamrozik, Konrad, Fraser, Gordon, Tillman, Nikolai, and Halleux, Jonathan. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.
- [10] Java PathFinder Tool-set. <http://babelfish.arc.nasa.gov/trac/jpf>.
- [11] Khurshid, Sarfraz, Păsăreanu, Corina S., and Visser, Willem. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.
- [12] King, James C. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.
- [13] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, November 2001.
- [14] Păsăreanu, Corina S. and Rungta, Neha. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.
- [15] Păsăreanu, Corina S., Mehrlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen, Lowry, Michael, Person, Suzette, and Pape, Mark. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.
- [16] Sen, Koushik and Agha, Gul. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.
- [17] Sinha, Saurabh, Shah, Hina, Görg, Carsten, Jiang, Shujuan, Kim, Mijung, and Harrold, Mary Jean. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 153–164, New York, NY, USA, 2009. ACM.
- [18] Song, JaeSeung, Ma, Tiejun, Cadar, Cristian, and Pietzuch, Peter. Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, 2011.

- [19] Souza, Matheus, Borges, Mateus, d'Amorim, Marcelo, and Păsăreanu, Corina S. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods, NFM'11*, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.
- [20] Tassey, G. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.
- [21] Tillmann, Nikolai and De Halleux, Jonathan. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs, TAP'08*, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.
- [22] von Detten, Markus. Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, PASTE '11*, pages 17–20, New York, NY, USA, 2011. ACM.
- [23] Weimer, Westley and Necula, George C. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04*, pages 419–431, New York, NY, USA, 2004. ACM.

VOSD: A General-Purpose Virtual Observatory over Semantic Databases*

Gergő Gombos[†], Tamás Matuszka[†], Balázs Pinczel[†],
Gábor Rácz[†] and Attila Kiss[†]

Abstract

E-Science relies heavily on manipulating massive amounts of data for research purposes. Researchers should be able to contribute their own data and methods, thus making their results accessible and reproducible by others worldwide. They need an environment which they can use anytime and anywhere to perform data-intensive computations. Virtual observatories serve this purpose. With the advance of the Semantic Web, more and more data is available in Resource Description Framework based databases. It is often desirable to have the ability to link data from local sources to these public data sets. We present a prototype system, which satisfies the requirements of a virtual observatory over semantic databases, such as user roles, data import, query execution, visualization, exporting result, etc. The system has special features which facilitate working with semantic data: visual query editor, use of ontologies, knowledge inference, querying remote endpoints, linking remote data with local data, extracting data from web pages.

Keywords: virtual observatory, semantic web, e-Science, data sharing, linked data

1 Introduction

E-Science is based on the interconnection of enormous amounts of data collected from various scientific fields. These massive data sets can be used for conducting researches, during which it is often desirable that researchers can share their own data and methods, thus making the results of the research accessible and reproducible by anyone. The idea of virtual observatories coming from Jim Gray and Alex S. Szalay serves this purpose [8]. A system like this expands the possibilities of combining data coming from various different instruments. Virtual observatories

*This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013). We are grateful to Zsófia Mészáros and Zoltán Vincellér for helpful discussion and comments.

[†]Eötvös Loránd University, Budapest, Hungary, E-mail: {ggombos, tomintt, vic, gabee33, kiss}@inf.elte.hu

can also be used to teach and demonstrate the basic research principles of various scientific fields (for example, astronomy or computer science). The researchers must have access to these constantly growing amounts of data, in order to be able to use them in various research projects. Another important requirement is to be able to publish the results. The Internet provides an excellent opportunity to satisfy the criteria mentioned above [8]. The primary motivation for creating virtual observatories is to facilitate making new discoveries, and to provide a solution for carrying out data-intensive computations remotely. To access remote data, web services can be used [19].

The basic principles of science have been extended with a fourth paradigm. A thousand years ago, experimental results and observations defined science. In the last few hundred years, it shifted towards a theoretical approach, focusing on creating and generalizing models. During the last few decades, simulating complex phenomena with computers were becoming more and more common. Nowadays, researchers have to deal with large amounts of data, usually coming from sensors, telescopes, particle accelerators, etc. The data is processed using software solutions, and the extracted knowledge is stored in databases. Analyzing or visualizing the results needs further software support [7, 11].

A possible way to manage the data available on the Internet is to use the Semantic Web [4]. The Semantic Web aims for creating a "web of data": a large distributed knowledge base, which contains the information of the World Wide Web in a format which is directly interpretable by computers. The goal of this web of linked data is to allow better, more sensible methods for information search, and knowledge inference. To achieve this, the Semantic Web provides a data model and its query language. The data model called the Resource Description Framework (RDF) [14] uses a simple conceptual description of the information: we represent our knowledge as statements in the form of subject-predicate-object (or entity-attribute-value) triples. This way our data can be seen as a directed graph, where a statement is an edge labeled with the predicate, pointing from the subject's node to the object's node. The query language called SPARQL [17] formulates the queries as graph patterns, thus the query results can be calculated by matching the pattern against the data graph. Furthermore, there are numerous databases which contain theoretical and experimental results of various scientific experiments in the field of computer science, biology, chemistry, etc. There is a quite complex collection of these kinds of data maintained by the Linked Data Community [5]. This collection contains datasets and ontologies which are at least 1000 lines in length, and which contain links to each other.

In this paper, we present a prototype system, which fulfills the standard requirements of a virtual observatory, such as handling user roles, bulk loading data, answering queries, visualization, and storing results. In addition, we extended the system with special semantic technologies. We use the SPARQL language to formulate queries, aided by a visual SPARQL editor. Ontologies can be used to describe the hierarchy of complex conceptual systems, and to carry out knowledge inference. The system implements a tool, which helps its users to convert the data found on the web to the formats of the Semantic Web. We also provide a SPARQL endpoint

to enable remote querying of the knowledge base. The query results can be exported to various common semantic data formats. We demonstrated the flexibility of the system by implementing two different database backends.

The structure of the paper is as follows. After the introductory Section 1, we outline preliminaries in Section 2. Afterwards, we present the high-level architecture of our virtual observatory in Section 3. Then, in Section 4, we describe the main functionality of the system. Then, we show some possible use cases of our system in Section 5, followed by the conclusion and our future plans in Section 6.

2 Preliminaries

As we mentioned in the introduction, the Semantic Web [4] provides various techniques to manage the data available on the Internet. This section gives insight into the basic concepts of Semantic Web that are necessary for understanding what our system is capable of and how it works. The main technologies that are used in our system are the following: Resource Description Framework (RDF), RDF Schema (RDFS), SPARQL query language, Web Ontology Language (OWL). In the formal discussion we follow the concepts and notations introduced in [16].

The Resource Description Framework is a description language, where the information is represented by RDF triples. Informally an RDF triple consists of a subject, a predicate, and an object; or alternatively it consists of an entity, a property, and the value of that property of the described entity. This representation form is similar to natural language sentences. For example the sentence '*Eötvös Loránd University is located in Budapest.*' can be translated into the triple (*Eötvös Loránd University*, *location*, *Budapest*). Three kinds of terms are distinguished: IRIs represent entities (e.g. <http://dbpedia.org/resource/ELTE>) or relations (e.g. <http://dbpedia.org/ontology/location>); literals can only occur as value of a property; blank nodes are the terms that do not represent real world entities, they just help to construct complex values, for example, mail addresses which consist of multiple parts such as postal code, city, street and number. Below is the formal definition of RDF triples (Definition 1).

Definition 1. Let I , B , and L (IRIs, Blank Nodes, Literals) be pairwise disjoint sets. An RDF triple is a $(v_1, v_2, v_3) \in (I \cup B) \times I \times (I \cup B \cup L)$, where v_1 is the subject, v_2 is the predicate and v_3 is the object. A finite set of RDF triples is called an RDF graph or RDF dataset.

The RDF Schema is a data-modeling vocabulary built on the top of RDF for defining concepts, properties and constraints which are essential for organizing the knowledge represented by triples. The Web Ontology Language also enables us to define concept and property hierarchies, however, it is a computational logic-based language. Therefore logical constraints and rules can be expressed in order to verify the consistency of that knowledge or to make implicit knowledge explicit. The formal definition of an ontology is presented in Definition 2, based on [20].

Definition 2. An ontology is a structure $\mathcal{O} := (C, \leq_C, P, \sigma)$, where C and P are two disjoint sets. The elements of C and P are called classes and properties, respectively. A partial order \leq_C on C is called class hierarchy and a function $\sigma: P \rightarrow C \times C$ is a signature of a property. For a property $p \in P$, its domain and its range can be defined in the following: $\text{dom}(p) := \pi_1(\sigma(p))$ and $\text{range}(p) := \pi_2(\sigma(p))$, where π is the projection operation. Let $c_1, c_2 \in C$ be two classes; if $c_1 \leq_C c_2$, then c_1 is a subclass of c_2 and c_2 is a superclass of c_1 .

SPARQL is a query language for retrieving and manipulating RDF data. It is an SQL-like declarative language; the queries are based on pattern matching, where the patterns are in the form of triples, though they can contain variables as well. Most of the keywords and their meanings are the same, such as *SELECT*, *WHERE*, *LIMIT*. However, there are some new keywords in SPARQL, for example, *OPTIONAL* means optional pattern matching, or *FILTER* that defines constraints for the variables. Definition 3 gives the abstract syntax of the filter conditions and Definition 4 presents the abstract syntax of the SPARQL expressions.

Definition 3. Let V be the set of distinct variables over $(I \cup B \cup L)$. The variables are distinguished by a question mark. Let $?X, ?Y \in V$ be variables and $c, d \in (L \cup I)$ be a literal and an IRI constant, respectively. We define the filter conditions recursively as follows. The $?X = c$, $?X = ?Y$, $c = d$, $\text{bound}(?X)$, $\text{isIRI}(?X)$, $\text{isLiteral}(?X)$, and $\text{isBlank}(?X)$ are atomic filter conditions. Thereafter, if R_1, R_2 are filter conditions, then $\neg R_1$, $R_1 \wedge R_2$ and $R_1 \vee R_2$ are filter conditions as well.

Definition 4. A SPARQL expression is built up recursively in the following way:

1. the triple $t \in (I \cup V) \times (I \cup V) \times (L \cup I \cup V)$ is a SPARQL expression,
2. if Q_1, Q_2 are SPARQL expressions, and R is a filter condition, then $Q_1 \text{ FILTER } R$, $Q_1 \text{ UNION } Q_2$, $Q_1 \text{ OPT } Q_2$, and $Q_1 \text{ AND } Q_2$ are SPARQL expressions as well.

The discussion of formal semantics of SPARQL is out of the scope of this paper. Set and multiset semantics are described in [16].

3 Architecture of the Virtual Observatory over Semantic Databases

The VOSD system was built using the Java EE platform. Figure 3 summarizes the main architectural elements of the system. As typical Java EE applications, VOSD consists of three major parts: a frontend, a business logic, and a database layer. Frontend is an interface for users, while backend contains the business logic which operates over the database.

As the Figure 3 shows, the basis of the system is an application server which is an Oracle WebLogic Application Server in our case. On the frontend side, our system uses the Java Server Faces (JSF) technology, which is a complete framework

for Java EE. This framework contains some basic elements, such as text boxes, message bars or pageable dynamic tables. In addition, it can handle file upload, even multiple files at once, error messages, user interactions. As JSF pages are supported by web browsers, the clients of the system can be various devices, for example, mobile phones, tablets, laptops. Besides the JSF pages, the system is also available via a REST webservice, to access the uploaded semantic models. This makes it possible to build different kind of applications over the system as you can see in Section 5 or in [13].

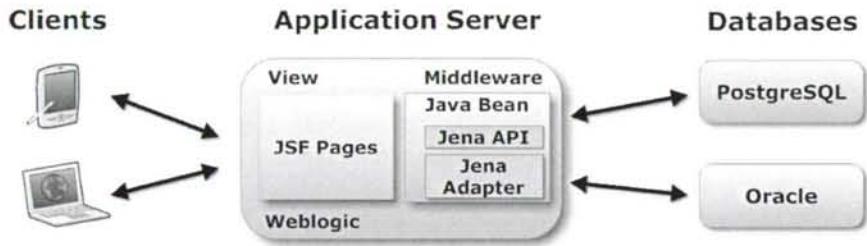


Figure 1: The architecture of the Virtual Observatory over Semantic Databases

On the backend side, two different databases are available by default. The first one is an Oracle 11g R2 database which supports the managing of semantic models and provides a Jena Adapter API for Java applications to use these features. Using the built-in semantic support, we can, for example, perform knowledge inference at the database level which can be much more faster than using a third-party tool. The second database is the PostgreSQL, which is a widely-used open-source relational database, however, it has no built-in semantic support. We chose this one to demonstrate how the already existing technologies can be applied to handle semantic data, and how efficient these two different solutions can be. On the top of PostrgesQL, Jena is used to map RDF data model to the relational model.

4 Functionality

In this section, we present the main functions of our system. Users can upload their own data sets in various formats. Then they can browse and query the uploaded datasets. A visual query editor is provided to facilitate the construction of syntactically correct SPARQL queries and the queries can be saved and re-used. Two third-party visualizer tools are integrated into our system to help understand and explore the structure of data sets. In addition, we offer a tool, which is able to extract RDF triples from semi-structured web pages. Last but not least, to support the collaboration of researchers, our system provides user group management. Users can share their own data sets and their own saved queries within groups or they can make their work publicly available for every user.

4.1 Data Loading

There are two ways to load data into the system. One works by uploading a file containing the semantic data, the other requires a URL pointing to a resource on the Internet which contains the data. There are various RDF serialization formats for RDF which can be used with the system, such as RDF/XML, N3, Turtle, and N-Triples. The most wide-spread is the RDF/XML, which represents the RDF graph as an XML document. This format is easier for computers to read, since there are numerous tools available for processing and transforming XML. The other formats store the data using a more human-readable serialization. The simplest one is the N-Triples [1], which is simply the enumeration of the RDF triples (the edges of the RDF graph) separated with a dot. The Turtle [2] serialization allows more structures to simplify the expressions. For example, we can use prefix abbreviations to eliminate long, repeating IRIs, thus reducing the file size significantly. Furthermore, we have the option to group triples sharing the same subject, without repeating the common subject for all triples. This works similarly, if both the subject and the predicates are the same, and only the objects vary. This, too, helps to reduce the file size. Literals in Turtle can have language tags, or data type information added to them. Notation 3 [3] (or N3) allows further simplifications to make the serialization of complex statements easier.

4.2 Querying and Saving Results

Another main function of the system is querying the already loaded data. The SPARQL [17] language is used to express queries over semantic data sets. The language is similar to the well-known SQL language. The (*SELECT*) clause defines a projection of the variables, the values for which we would like to see in the result set. The *WHERE* clause defines the criteria the data must satisfy in order to appear as a result. This is basically a graph pattern that has to match the data graph. The simplest queries contain only triples in the graph pattern. The *FILTER* clause lets us provide further filtering conditions for the nodes. For example, if we have numeric nodes, we can use arithmetic operators on them to restrict the values to a given range. If we have string nodes, we can filter for their values as well. IRIs, and string nodes can be filtered using regular expressions, too. By default, all edges in the graph pattern of the *WHERE* clause have to match the data. However, we have the option to define optional matching criteria with the *OPTIONAL* keyword. If parts of the graph pattern are optional, then we can have rows in the result set which satisfy only the non-optional parts, with null values for the variables appearing only in the optional parts. This is useful when some information is not given for all of our individuals. For example, if we have an address book with addresses for all contacts and phone numbers for some of them, we can ask the phone numbers in the optional part. Without the *OPTIONAL* keyword, we would only get the contacts with both an address and a phone number.

The advantage of the Semantic Web is that we can link our data with knowledge from other sources. In queries, the *SERVICE* keyword allows querying remote data

sets. The keyword requires a URL to a SPARQL endpoint, and a graph pattern that has to match the remote data. The most well-known data set is the DBpedia [6], which contains a subset of the knowledge of Wikipedia in semantic form. Data sets linked with DBpedia can be found in the LOD cloud [5].

Another useful feature of the semantic web is knowledge inference, which lets us extract new information based on what we already know. Computing inferred data may take long time, thats why our system offers two options regarding inference. One option is to run the query using only the data already available to us as facts, or we can enable inference – meaning slower query execution. There are multiple ways to carry out inference. For example, we can use the relationship information given in ontologies to generate new information. Another option is to use user-specified rules. A rule consists of a head (a new triple holding the new information) and a body (a condition that has to be satisfied in order for the rule to activate). The simplest example is the grandparent relationship (if x is parent of y , and y is parent of z , then x is grandparent of z). We can save the query results using the already mentioned formats: RDF/XML, N3, TURTLE, and also CSV.

4.3 Visual SPARQL Editor

With the spreading of the Semantic Web technologies, using SPARQL becomes more and more inevitable, since this declarative language is the standard tool to express queries over RDF data sets. VisualQuery is a visual query editor program, which allows us to build a SPARQL query using graphs and supplementary forms.

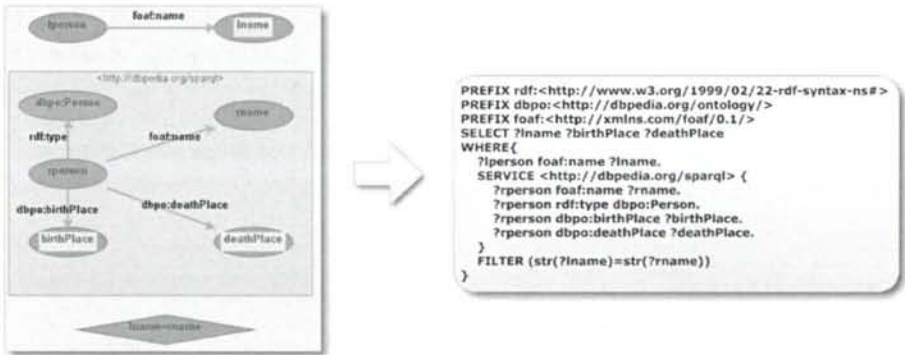


Figure 2: An example SPARQL query both in graphic and textual form which finds additional information on DBpedia about locally stored famous people

Graphic representation has various advantages. Firstly, using this approach, it is easier to see and understand the relationship of the individual elements, thus, the meaning of the query can clearly be seen as demonstrated in Figure 2 where the graphic and textual representation of the same query are shown. Secondly, we can quickly and easily modify the components and parameters defining the query. This way, we can improve or refine the query step-by-step. Thirdly, because the

visual representation is language-independent, the co-operative work of researchers speaking different languages is supported. Another advantage of the program is that it performs various checks during editing, which helps preventing syntactical errors, for example:

- literal nodes can not have outgoing edges – they can not be subjects in a triple,
- only variables or IRI nodes can be edges – blank nodes and literals can not,
- variables in the head of a CONSTRUCT-type query must appear at least once in the WHERE clause.

What makes this solution different from similar programs – like iSparql [15] or LuposDate [9] – is the distinction of visual elements by type, and the built-in checks based on this distinction.

4.4 Visualizations

We mentioned earlier that the semantic data can be seen as a directed graph. Subjects and objects are the nodes, and the predicates are the edges of the graph. Visualizing this graph helps us interpret the data. More graph visualization tools are available, and some can visualize the semantic data. We integrated two third-party visualizer tools into the system, that is seen on Figure 3.

One of them is Cytoscape Web [18], which allows us to display the semantic graph of locally stored models using various built-in layouts, such as a tree or circle. It is an open-source, interactive, customizable tool. It is a simple version of the Cytoscape for the web and it is reusable. The application uses Flex/Actionscript with JavaScript API, so rendering happens on the clients' computer. The user can visualize own models and the public models. However, the models can contain large amounts of data, and visualizing these models are resource-intensive, so we have to limit the edges.

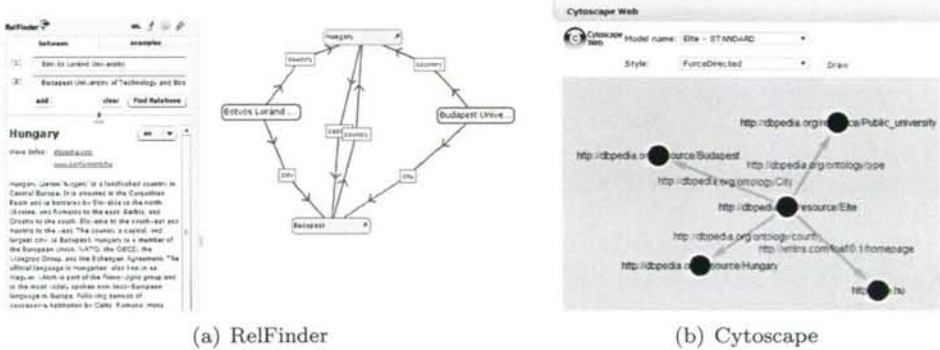


Figure 3: Visualization tools

Another visualization tool integrated into the system is RelFinder [10], which searches connections among IRIs. To find connections, it runs SPARQL queries on an endpoint. The relations among the IRIs can be paths via common predicates. RelFinder first finds the shortest path, and adds its nodes to the graph. After that it tries to find longer paths. We can specify the maximum depth of the search. The program uses Flex/ActionScript for the display that provides various tools to create animations. We configured this tool to work on the semantic data of the virtual observatory, and the users can search their own models.

4.5 Extracting Semantic Data from the Web

Nowadays, we can easily find all kinds of information using the web. There are numerous sites which specialize in collecting and organizing knowledge about one specific topic. For example, we can find websites collecting information about hardware components, reviews about movies, historical weather data, recipe collection, etc. These websites usually operate using a database of their own, and the web pages displayed to us are generated dynamically using the stored data. However, the databases are usually not using semantic technologies, moreover, they are often not public, so the only way for us to access the data is to visit the web pages. Fortunately, extracting data from the web pages does not always require complex text processing and text mining, because the consistent structure of the documents can be utilized to extract the information we are interested in. The structure is almost always consistent on all pages of a web site. For example, on a site collecting recipes, the structure can be the following: the name of the dish is always the title of a section, and it is followed by some additional information (always in the same order), such as the name of the uploader, the difficulty and the required time to prepare the meal. After this, we have a bullet-point list of the ingredients, and finally, there is a numbered list of the steps in the recipe. If we know this structure, we can utilize it to extract all recipes from all pages of the site.

To help users in extracting data from sites like these, we created a browser extension that allows them to define the structure using one example page of a web site. Based on the structure information created this way, our virtual observatory is capable to extract the required information from all pages that use the same document structure. The tool can be downloaded and installed from the web front end of our virtual observatory. Then, visiting the desired website, the user can mark the sections to be extracted using selection with the mouse. To extract information about all entities (e.g. all recipes) on the same page, the user only needs to annotate the first occurrence by binding variables to the parts of interest (e.g. the name of the dish and the list of ingredients). These variables can be used to formulate an RDF template, which – during the extraction phase – gets instantiated for each entity on all similar-structured pages of the website, with the appropriate extracted values in place of the variables.

The inner model for the annotation and the extraction is based on the DOM (Document Object Model) tree of the web page. When the user marks the first occurrence of an entity for extraction, the corresponding node in the DOM tree is

marked as an anchor. Variables are defined relative to an anchor, by the (possibly empty) path that leads from the anchor to the node bound to the variable. During the extraction phase, occurrences of the repeating structure will be traversed by iterating over those sibling nodes of the anchor which have the same type (i.e. HTML tag). In each iteration, the appropriate variables are evaluated by following their defined path, starting from the current sibling. To handle repeating structure inside a repeating structure (e.g. ingredients as list items inside recipes), the anchors can be nested, resulting in a nested iteration during the extraction phase. Figure 4 shows an example model with two anchors (one of them nested), and two variables.

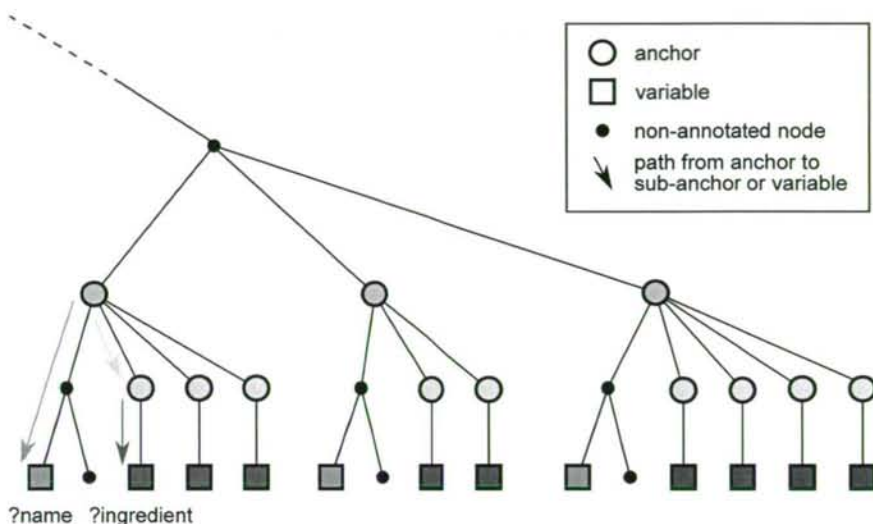


Figure 4: Part of an annotated DOM tree with two anchors and two variables.

The model described in the previous paragraph is automatically created and updated in the background, whenever the user marks an area for extraction or binds a variable to an element on the page. This way, no knowledge of DOM trees and paths are required to use the tool: the model for the data extraction can be created in a user-friendly way, using selection by mouse and a few clicks in the drop-down menu of the browser extension. The created model is saved as an XML file, which contains the structure information (anchors, variables, and paths between them) and the RDF template. The virtual observatory takes this file and a list of URLs as input, extracts the information from the specified web pages, and saves the extracted data to a semantic model.

4.6 Collaboration of Researchers

One of the most important purposes for virtual observatories is to collect information originating from various different sources, and to support their integration.

Our system allows users to upload their own data and share it with others. We applied a multi-level permission system based on user groups. Every user can create groups, and invite other users to them. This way, research groups can be organized. Then, we have two possibilities to share the models containing our data. We can make the model publicly available to every other user, or we can give right to one or more groups to access our model. While the first possibility gives read-only access, in the latter case the group members can have write rights, too. In this case, they can load their own data into the model.

It is also possible to publish queries. This can be useful in several cases: if other researchers would like to use our data, we can help their work by providing example queries, which illustrate the inner structure and relationships of the data. We can formulate basic queries, which can be further refined or specialized later.

5 Use Cases

In this section we describe two use cases that show the advantages of our system. The first one sums up how an application can be built on the top of data that is collected from heterogeneous sources, and how our system can be used to develop and manage such applications. The second example presents how we use the system in the education, how the functions and tools help the students to get familiar with the basic principles of the Semantic Web.

5.1 OCR Application

The first application is useful in the field of tourism. The main function of the program is to recognize text on street signs with OCR methods, based on pictures taken with mobile phones. Its purpose is to provide extra information about the famous people whose name can be found in the extracted texts. The extra information comes from various data sources converted to semantic format (Hungarian Electronic Library, various online encyclopedias [12]), joined with other public data sets (DBpedia, GeoNames). A user group created for this purpose allows the collaboration between the users. The group has access to the data sets described above. One member of the group was given the task to collect information about the famous people appearing in street names, and then upload them to a model. He then shared the model inside the group. Another member had the same task, but he had to use an online encyclopedia as the data source. He added his data to the shared model. Meanwhile, a third member worked on linking the data in the model to data available in DBpedia, using SPARQL queries. He stored the results in a new, local model, to make it faster to access. (His work was not influenced by the fact that in the meantime, new data has been added to the model.) He also published the queries and the new model to the group. The members of the group created a virtual model over the models mentioned. (A virtual model is not materialized, but it contains the union of the data found in other models, and it is supported by an index structure.) This step was important, because it allowed

us to access the data as a single model. Then, using the REST API of our virtual observatory, we were able to run queries from a mobile application.

5.2 Use in Education

We use the virtual observatory during teaching the basic principles of the Semantic Web, within the Modern Databases course. The students of the course are added to a new group, and we share previously loaded models and queries with them. The models contain small data sets, so they could be viewed with the visualization tools, and the students could easily understand their structure. From week to week, they are introduced to the features of the SPARQL language, by solving typical tasks together. The new features can easily be demonstrated with the visual SPARQL editor, since the graphical representation speaks for itself. In some cases, the results of the exercises can be used in practical scenarios. For example, the family tree of a royal family can be created, if each student creates a model with the family tree of a selected king. During their work, they get to know the basic semantic serialization formats (RDF/XML, N3, etc.) and the results can be published to a common group.

6 Conclusion and Future Work

In the paper, we presented a prototype system, which fulfills the requirements of a virtual observatory, and helps the collaboration of researchers by letting them work using the same shared data and queries. We used the data model of the Semantic Web, thus the data sets in the virtual observatory can easily be linked to each other and to public data sets. We provided several features which can facilitate the use of the system, such as advanced data and query sharing, visual query building and editing, data visualization, and web data extraction. The system can run on top of any standard relational database system, but if the underlying database has some support for storing and handling semantic data (like Oracle databases), it can make use of those functions as well. We also presented real world use cases, where the existence of the system helped our work on other projects and in education. We are currently working on incorporating the ability to build and maintain bisimulation-based structure indexes, and utilize them in query evaluation to achieve better performance. Another feature in development is the visualization of SPARQL query plans. During further work, we would like to extend the system to be able to work using a Hadoop cluster as backend. In this solution, data storage and query execution would be distributed, thus the efficiency of the data-intensive computations would increase. Our other plans include enhanced visualization, such as the ability to plot geographic locations on a map, and to create charts and diagrams to help the better understanding of the data.

References

- [1] Beckett, D. RDF 1.1 N-triples. W3C Recommendation, 2014. <http://www.w3.org/TR/n-triples/>
- [2] Beckett, D., Berners-Lee, T., Prud'hommeaux, E. and Carothers, G. RDF 1.1 Turtle - Terse RDF Triple Language W3C Recommendation, 2014. <http://www.w3.org/TR/turtle/>
- [3] Berners-Lee, T. and Connolly, D. Notation3 (N3): A readable RDF syntax W3c Team Submission, 2011. <http://www.w3.org/TeamSubmission/n3/>
- [4] Berners-Lee, T., Hendler, J. and Lassila, O. The semantic web. *Scientific American*, 284(5): 28–37, 2001.
- [5] Bizer, C., Heath, T. and Berners-Lee, T. Linked data-the story so far. *International journal on semantic web and information systems* 5(3): 1–22, 2009.
- [6] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R. and Hellmann, S. DBpedia – A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web* 7(3): 154–165, 2009.
- [7] Brase, J. and Blümel, I. Information supply beyond text: non-textual information at the German National Library of Science and Technology (TIB) – challenges and planning. *Interlending & Document Supply*, 38(2): 108–117, 2010.
- [8] Gray, J. and Szalay, A. The world-wide telescope. *Communications of the ACM*, 45(11): 50–55, 2002.
- [9] Groppe, J., Groppe, S., Schleifer, A. and Linnemann, V. LuposDate: A semantic web database system. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2083–2084. ACM, 2009.
- [10] Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T. Relfinder: Revealing relationships in rdf knowledge bases. In *Semantic Multimedia*, pages 182–187. Springer, 2009.
- [11] Hey, T., Tansley, S. and Tolle, K. M. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, Redmond, 2009.
- [12] Hungarian Electronic Library <http://mek.oszk.hu/indexeng.phtml>
- [13] Gombos, G., Matuszka, T., Pinczel, B., Rácz, G., Kiss, A. and Gaizer, T. A Semantic Browser for Enterprise Information Systems on Mobile Platform In *Proceedings of the 12th International Scientific Conference on Informatics'2013*, pages 246–251. 2013.

- [14] Manola, F., Miller, E. and McBride, B. RDF 1.1 Primer. W3C Recommendation, 2014. <http://www.w3.org/TR/rdf11-primer/>
- [15] OAT Interactive SPARQL (iSPARQL) Query Builder <http://oat.openlinksw.com/isparql/index.html>
- [16] Pérez, J., Arenas, M. and Gutierrez, C. Semantics and Complexity of SPARQL. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.
- [17] Prud'hommeaux, E. SPARQL 1.1 Query Language W3C Recommendation, 2013. <http://www.w3.org/TR/sparql11-query/>
- [18] Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Ideker, T. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11): 2498–2504, 2003.
- [19] Szalay, A. S., Budavári, T., Malik, T., Gray, J. and Thakar, A. R. Web services for the virtual observatory. In *Astronomical Telescopes and Instrumentation*, pages 124–132. International Society for Optics and Photonics, 2002.
- [20] Volz, R., Kleb, J. and Mueller, W. Towards Ontology-based Disambiguation of Geographical Identifiers. In *I3*, 2007.

Monitoring Evolution of Code Complexity and Magnitude of Changes

Vard Antinyan*, Mirosław Staron*, Jörgen Hansson*,
Wilhelm Meding†, Per Österström‡ and Anders Henriksson‡

Abstract

Complexity management has become a crucial activity in continuous software development. While the overall perceived complexity of a product grows rather insignificantly, the small units, such as functions and files, can have noticeable complexity growth with every increment of product features. This kind of evolution triggers risks of escalating fault-proneness and deteriorating maintainability. The goal of this research was to develop a measurement system which enables effective monitoring of complexity evolution. An action research has been conducted in two large software development organizations. We have measured three complexity and two change properties of code for two large industrial products. The complexity growth has been measured for five consecutive releases of the products. Different patterns of growth have been identified and evaluated with software engineers in industry. The results show that monitoring cyclomatic complexity evolution of functions and number of revisions of files focuses the attention of designers to potentially problematic files and functions for manual assessment and improvement. A measurement system was developed at Ericsson to support the monitoring process.

Keywords: complexity, metrics, risk, lean, agile, correlation, measurement systems, code, change, revision

1 Introduction

Actively managing software complexity has become an important aspect of continuous software development. It is generally accepted that software products developed in a continuous manner are getting more and more complex over time. Evidence shows that the rising complexity drives to deteriorating quality of software [2,3]. The continuous increase of code base and growing complexity can lead to large, virtually unmaintainable source code if left unmanaged.

*Computer Science and Engineering, University of Gothenburg | Chalmers, E-mail: {vard.antinyan,mirosław.staron,jörgen.hansson}@chalmers.se

†Ericsson, E-mail: {wilhelm.meding,per.osterstrom}@ericsson.com

‡Volvo Group Truck Technology, E-mail: anders.J.henriksson@volvo.com

A number of metrics have been suggested to measure various aspects of software complexity and evolution over development time [7]. Those metrics has been accompanied with a number of studies indicating how adequately the proposed metrics relate to software quality [6, 17]. Complexity and change metrics have been used extensively in recent years for assessing the maintainability and fault-proneness of software code [4]. Despite the considerable amount of research conducted for investigating the influence of complexity on software quality, little results can be found on how to effectively monitor and prevent complexity growth. Therefore a question remains:

How to monitor code complexity and changes effectively when delivering feature increments to the main code branch?

The aim of this research was to develop method and tool support for actively monitoring complexity evolution and drawing the attention of industries' software engineers to the potentially problematic trends of growing complexity. In this paper we focus on the level of self-organized software development teams who often deliver code to the main branch for further testing, integration with hardware, and ultimate deployment to end customers. We address this question by conducting a case study at two companies, which develop software according to Agile and Lean principles. The studied companies are Ericsson which develops telecom products and Volvo Group Truck Technology (GTT) which develops electronic control units (ECU) for trucks.

Our results show that using two complementary measures, McCabes cyclomatic complexity of functions and number of revisions of files supports teams in decision making, when delivering code to the main branch. The evaluation shows that monitoring trends in these measures draws attention of the self-organized Agile teams to a handful of functions and files. These functions and files are manually assessed, and the team formulates decisions before the delivery on whether they can cause problems.

2 Related Work

Continuous software evolution: A set of measures useful in the context of continuous deployment can be found in the work of Fritz [8], in the context of market driven software development. The metrics presented by Fritz measure such aspects as continuous integration as pace of delivery of features to the customers. These metrics complement the two indicators presented in this paper with business perspective which is important for product management.

The delivery strategy, which is an extension of the concept of continuous deployment, has been found as one of the three key aspects important for Agile software development organizations in a survey of 109 companies by Chow and Cao [5]. The indicator presented in this paper is a means of supporting organizations in their transition towards achieving efficient delivery processes.

Ericssons realization of the Lean principles combined with Agile development was not the only one recognized in literature. Perera and Fernando [14] presented

another approach. In their work they show the difference between the traditional and Lean-Agile way of working. Based on our observations, the measures and their trends at Ericsson were similar to those observed by Perera and Fernando.

Measurement systems: The concept of an early warning measurement system is not new in engineering. Measurement instruments are one of the cornerstones of engineering. In this paper we only consider computerized measurement systems i.e. software products used as measurement systems. The reasons for this are: the flexibility of measurement systems, the fact that we work in the software field, and similarity of the problems e.g. concept of measurement errors, automation, etc. An example of a similar measurement system is presented by Wisell [21] where the concept of using multiple measurement instruments to define a measurement system is also used. Although differing in domains of applications these measurement systems show that concepts which we adopt from the international standards (like [11]) are successfully used in other engineering disciplines. We use the existing methods from the ISO standard to develop the measurement systems for monitoring complexity evolution.

Lowler and Kitchenham [12] present a generic way of modeling measures and building more advanced measures from less complex ones. Their work is linked to the TychoMetric [15] tool. The tool is a very powerful measurement system framework, which has many advanced features not present in our framework (e.g. advanced ways of combining metrics). A similar approach to the TychoMetrics way of using metrics was presented by Garcia et al. [9]. Despite their complexity, both the TychoMetric tool and Garcias approach can be seen as alternatives in the context of advanced data presentation or advanced statistical analysis over time. Our research is a complement to [13] and [15]. We contribute by showing how the minimal set of measures can be selected and how the measurement systems can be applied regularly in large software organizations.

Meyer [10, pp. 99-122] claims that the need for customized measurement systems for teams is one of the most important aspects in the adoption of metrics at the lowest levels in the organization. Meyers claims were also supported by the requirements that the customization of measurement systems and development of new ones should be simple and efficient in order to avoid unnecessary costs in development projects. In our research we simplify the ways of developing Key Performance Indicators exemplified by a 12-step model of Parmenter [13] in the domain of software development projects.

3 Design of the Study

This case study was conducted using action research approach [1,16]. The researchers were part of the companys operations and worked directly with product development units. The role of Ericsson in the study was the development of the method and its initial evaluation, whereas the role of Volvo GTT was to evaluate the method in a new context.

3.1 Studied Organizations

Ericsson: The organization and the project within Ericsson developed large products for mobile packet core network. The number of the developers in the projects was up to a few hundreds. Projects were executed according to the principles of Agile software development and Lean production system, referred to as Streamline development within Ericsson [20]. In this environment, different development teams were responsible for larger parts of the development process compared to traditional processes: design teams, network verification and integration, testing, etc.

Volvo GTT: The organization which we worked with at Volvo GTT developed ECU software for trucks. The collaborating unit developed software for two ECUs and consisted of over 40 designers, business analysts and testers at different levels. The development process was in the transition from traditional to Agile.

3.2 Units of Analysis

During our study we analyzed two different products: software for a telecom product at Ericsson and software for two ECUs at Volvo GTT.

Ericsson: The product was a large telecommunication product composed by over two million lines of code with several tens of thousands C functions. The product had a few releases per year with a number of service releases in-between them. The product has been in development for a number of years.

Volvo GTT: The product was an embedded software system serving as one of the main computer nodes for a product line of trucks. It consisted of a few hundred thousand lines of code and several thousand C functions. The analyses that were conducted at Ericsson were replicated at Volvo GTT under the same conditions and using the same tools. The results were communicated with designers of the software product after the data was analyzed. At Ericsson the developed measurement system ran regularly whereas at Volvo the analysis was done semi-automatically, that is, running the measurement system whenever feedback was needed for designers.

3.3 Reference Group

During this study we had the opportunity to work with a reference group at Ericsson and a designer at Volvo GTT. The aim of the reference group was to support the research team with expertise in the product domain and to validate the intermediate findings as prescribed by the principles of Action research. The group interacted with researchers on a bi-weekly meeting basis for over 8 months. At Ericsson the reference group consisted of a product manager, a measurement program leader, two designers, one operational architect and one research engineer. At Volvo GTT we worked with one designer.

3.4 Measures in the Study

Table 1 presents the complexity measures, change measures and deltas of complexity measures over time. The definitions of measures and their deltas are provided also.

Table 1: Metrics and their definitions

Complexity Measures	Abbrev.	Definition
McCab's cyclomatic complexity of a function	M	The number of linearly independent paths in the control flow graph of a function, measured by calculating the number of "if", "while", "for", "switch", "break", "&&", " " tokens
Structural <i>Fan-out</i>	<i>Fan-out</i>	The number of invocations of functions found in a specified function
Maximum Block Depth	MBD	The maximum level of nesting found in a function
Cyclomatic complexity of a file	M_f	The sum of all functions M in a file
Change Measures	Abbrev.	Definition
Number of revisions of a file	NR	The number of check-ins of files in a specified code integration branch and its all sub-branches in a specified time interval
Number of designers of a file	ND	The number of developers that do check-in of a file on a specified code integration branch and all of its sub-branches during a specified time interval
Deltas of Complexity Measures	Abbrev.	Definition
Complexity deltas of a function	$\Delta M,$ $\Delta Fan-out,$ ΔMBD	The increase or decrease of M , <i>Fan-out</i> and MBD measures of a function during a specified time interval.

3.5 Research Method

According to the principles of action research we adjusted the process of our research with the operations of the company. We conducted the study according to the following pre-defined process:

- Obtain access to the source code of the products and their different releases
- Calculate complexity measures of all functions and change measures of all files in the code
- Calculate the complexity deltas of all functions through five releases of both products
- Sort the functions by complexity delta through five releases
- Identify possible patterns of complexity change
- Identify drivers for complexity changes for functions with functions having highest overall delta
- Correlate measures to explore their dependencies and select measures for monitoring complexity and changes
- Develop a measurement system (according to ISO 15939) for monitoring complexity and changes
- Monitor and evaluate the measurement system for five weeks

The overall complexity change of function is calculated by:

$$\text{Overall delta} = |\Delta M_{1-2}| + |\Delta M_{2-3}| + |\Delta M_{3-4}| + |\Delta M_{4-5}|.$$

$|\Delta M_{i-j}|$ is the absolute value of change of M of a function between i and j releases. Overall complexity change of Fan-out and MBD is calculated the same way.

4 Analysis and Results

In this section we explore the main scenarios of complexity evolution. We carry out correlation analysis of collected measures in order to understand their dependencies and chose measures for monitoring.

4.1 Evolution of the Studied Measures Over Time

Exploring different types of changes of complexity, we categorized changes into 5 groups.

1. Group 1 - Functions that are newly created and become complex in current release and functions that existed but disappeared in current release.
2. Group 2 - Functions that are re-implemented in current release.
3. Group 3 - Functions that have significant change of complexity between two releases due to development or maintenance.

- 4. Group 4 - Test functions, which are regularly generated, destroyed and re-generated for unit testing.
- 5. Group 5 - functions that have minor complexity changes between two releases.

Group 1 and group 5 functions were observed to be the most common. They appeared regularly in every release. Engineers of the reference group characterized their existence as expected result of software evolution. Group 2 functions were re-implementation of already existing function. The existed functions were re-implemented with different name and the old one was destroyed. After re-implementation the new functions could be named as the old one. Re-implementation usually took place when major software changes were happening: In this case re-implementation of a function sometimes could be more efficient than modification. Figure 1 shows the cyclomatic complexity evolution of top 200 functions through five releases of products. Each line on the figure represents a C function.

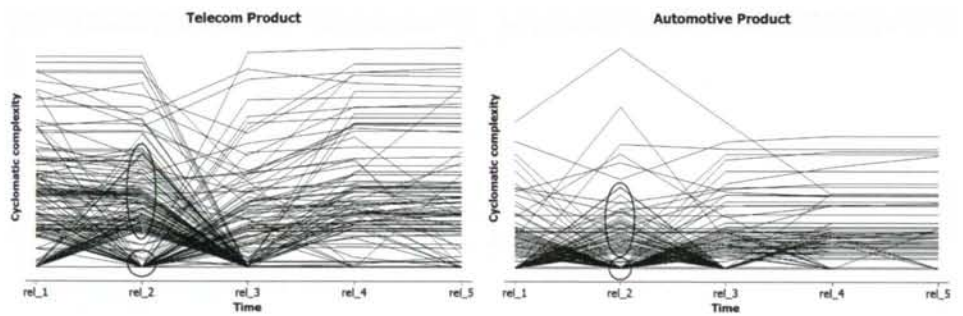


Figure 1: Evolution of M of functions

In Figure 1 re-implemented functions are outlined by elliptic and old ones by round lines. In reality the number of re-implemented functions are small (about 1%), however considering the big magnitude of complexity change of them, many of them ended-up in the top 200 functions in the picture, giving an impression that they are relatively many. Figure 2 similarly presents the evolution of Fan-out in the products. Group 3 functions are outlined by elliptic line in Figure 2.

Group 3 functions were usually designed for parsing a huge amount of data and translating them into another format. As the amount and type of data is changed the complexity of the function also changes. Finally the Group 5 functions were unit test implementations. These functions were destroyed and regenerated frequently in order to update running unit tests. Figure 3 presents the MBD evolution of products. As nesting depth of blocks can be relatively shallow, many lines in Figure 3 overlap each other thus creating an impression that there are few functions. We observed that functions in group 1, ones were created, stayed complex over time. These functions are outlined with a rectangular line in Figure 3.

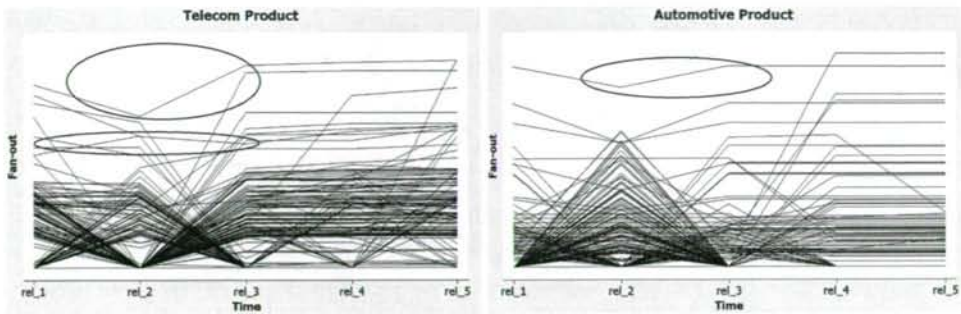


Figure 2: Evolution of Fan-out of functions

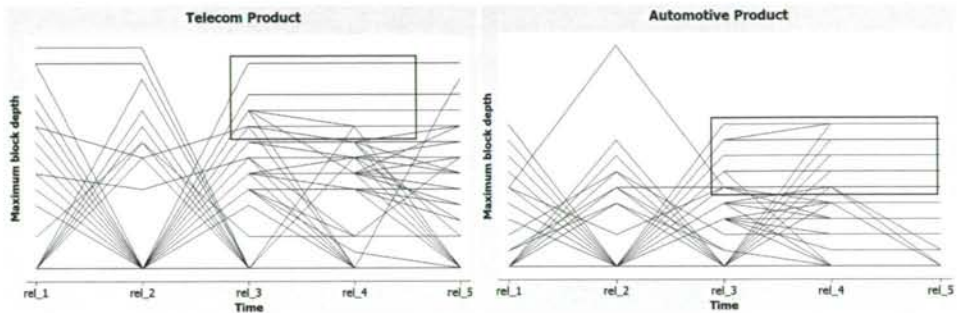


Figure 3: Evolution of MBD of functions

The statistics of functions of all groups are represented in Table 2. The table shows how all functions, that had complexity change, are distributed in groups. We would like to mention that the number of all functions in telecom product is about 65000 and in automotive product about 10000, however only top 200 functions out of those are presented in the figures. This might result in disproportional visual relationship between the relation of different groups of functions in the table and in the figures as the figures contains only top 200 functions.

Table 2: The distribution of functions with complexity delta in groups

Group	Group 1	Group 2	Group 3	Group 4	Group 5
Percentage	27%	1%	1%	1%	70%

We observed the change of complexity for both long time intervals (between releases) and for short time intervals (between weeks). Figure 4 shows how the complexity of functions changes over weeks. The initial complexity of functions is provided under column M in the figure. We can see the week numbers on the

top of the columns, and every column shows the complexity growth of functions in that particular week. Under column we can see the overall delta complexity per function that is the sum of weekly deltas per function.

File name	Function name	M	Total: ΔM	w1306	w1307	w1308	w1309	w1310	w1311	w1312
file 1	function 1	14	0	0	0	0	0	0	0	0
file 2	function 2	15	15	0	0	0	0	0	15	0
file 2	function 3	1	0	0	0	0	0	0	0	0
file 3	function 4	10	5	4	-9	11	-11	10	0	0
file 4	function 5	11	3	0	0	0	0	11	0	0
file 5	function 6	58	13	17	0	11	-11	0	0	-4
file 5	function 7	22	22	0	0	0	0	0	0	22
file 6	function 8	20	20	0	0	0	18	2	0	0
file 6	function 9	17	17	0	0	0	17	0	0	0
file 7	function 10	11	11	0	0	0	11	0	0	0
file 8	function 11	13	13	0	0	0	0	13	0	0
file 9	function 12	28	28	0	28	0	0	0	0	0
file 10	function 13	12	12	0	0	0	12	0	0	0

Figure 4: Visualizing complexity evolution of functions over weeks

The fact that the complexity of functions fluctuates irregularly was interesting for the designers, as the fluctuations indicate active modifications of functions, which might be due to new feature development or represent defect removals with multiple test-modify-test cycles. Functions 4 and 6 are such instances illustrated in Figure 4. Monitoring the complexity evolution through short time intervals we observed that very few functions are having significant complexity increase. For example in a week period the number of functions that have complexity increase $\Delta M > 10$ can vary between 5-10 while overall number of functions reaches a few tens of thousands in the product.

4.2 Correlation analyses

The correlation analyses of measures were conducted in order to eliminate dependent measures and select a minimal amount of measures for monitoring. The correlation analysis results of complexity measures for the two software products are presented in Table 3. The visual presentation of the relationship of complexity measures is presented in Figure 5. As the table illustrates there is a strong correlation between M and Fan-out for the telecom product and M and MBD for the automotive product. There is a moderate correlation between M and MBD for the telecom product. Generally designers of reference group concluded that monitoring the cyclomatic complexity among all complexity measures is good enough as there was a moderate or strong correlation between three complexity measures. M was chosen because of two reasons:

- 1. MBD is rather a characteristic of a block of code than a whole function. It is a good complementary measure but it cannot characterize the complexity of a whole function.
- 2. Fan-out seemed to be a weaker indicator of complexity than M because it rather showed the vulnerability of a function towards other functions that are in that function.

Considering aforementioned conclusions M was chosen among complexity measures to be monitored.

Table 3: Correlation of complexity measures

Telecom / Automotive	MBD	M
M	0.41 / 0.69	
Fan-out	0.34 / 0.20	0.76 / 0.26

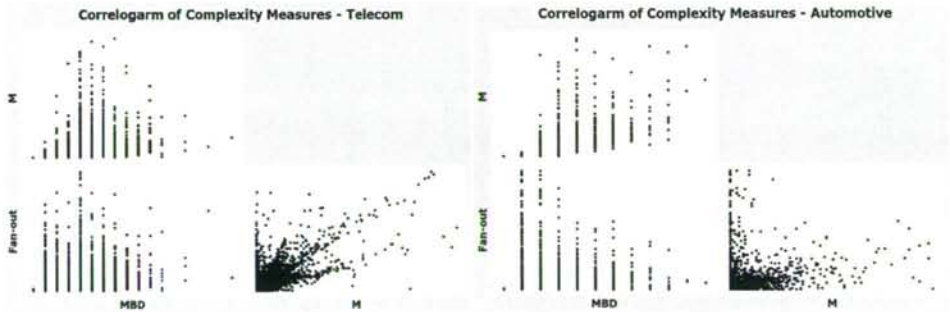


Figure 5: Correlogram of complexity measures

NR and ND are measures that indicate the magnitude of changes. Previously a few studies have shown that change metrics are good indicators of problematic areas of code, as observed Shihab [18]. The measurement entity of NR and ND is a file. Therefore in order to understand how change measures correlates to complexity we decided to define the M measure for files (Table 1). Table 4 presents the correlation analysis results for ND, NR and M_f measures.

An important observation was the strong correlation between the number of designers and the number of revisions for the telecom product (Table 4). At the beginning of this study the designers of the reference group at Ericsson believed that a developer of a file might check-in and check-out the file several times which probably is not a problem. The real problem, they thought, could be when many designers modify a file simultaneously. Nonetheless, a strong correlation between the two measures showed that they are strongly dependent, and many revisions

is mainly caused by many designers modifying a file in a specified time interval (Figure 6).

Table 4: Correlation of change and complexity measures

Telecom / Automotive	M_f	ND
ND	0.40 / 0.37	
NR	0.46 / 0.72	0.92 / 0.41

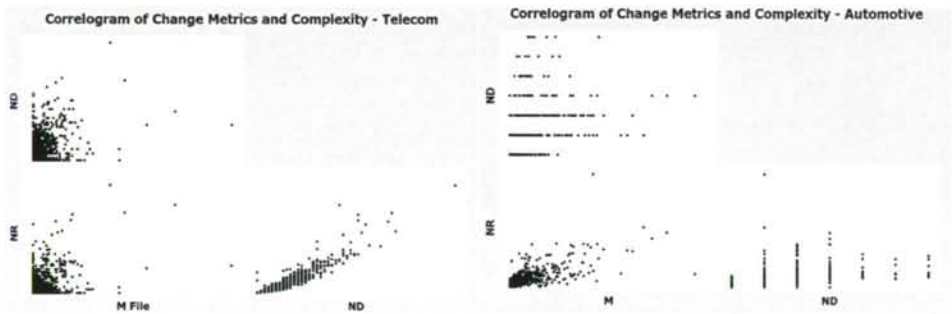


Figure 6: Correlogram of change and complexity measures

In case of automotive product correlation of ND and NR was moderate which can be due to small number of designers who have rather firmly assigned development areas and usually change the same code. Moderate correlation between M_f and NR for the telecom product indicates that complex files are prone to changes. There are always simple files that are changed often due to development.

Considering the correlation analysis results we designed a measurement system at Ericsson for monitoring code complexity and magnitude of changes over time. The description of design and application of measurement system is discussed in the next section.

4.3 Design of the Measurement System

Based on the results that we obtained from investigation of complexity evolution and correlation analyses, we designed two indicators based on M and NR measures. These indicators capture the increase of functions complexity and highlight the files with highest change magnitude over time. These indicators were designed according to ISO/IEC 15959. The design of complexity indicator is presented in Table 5. The other indicator based on NR is defined in the same way: the files that had $NR > 20$ during last week development time period should be identified and reviewed. The measurement system was provided as a gadget with the necessary information updated on a weekly basis (Figure 7). The measurement system relies on a previous

study carried out at Ericsson [19]. For instance the total number of files with more than 20 revisions since last week is 5 (Figure 7). The gadget provides the link to the source file where the designers can find the list of files or functions and the color-coded tables with details.

We visualized the NR and ΔM measures using tables as depicted in Figure 4. As in Streamline development the development team merged builds to the main code branch in every week it was important for the team to be notified about functions with drastically increased complexity (over 20).

Table 5: Measurement system design based on ISO/IEC 15939 standard

Information Need	Monitor cyclomatic complexity evolution over development time
Measurable Concept	Complexity change of delivered source code
Entity	Source code function
Attribute	Complexity of C functions
Base Measures	Cyclomatic complexity number of C functions M
Measurement Method	Count cyclomatic number per C function according to the algorithm in CCCC tool
Type of measurement method	Objective
Scale	Positive integers
Unit of measurement	Execution paths over the C/C++ function
Derived Measure	The growth of cyclomatic complexity number of a C function in one week development time period
Measurement Function	Subtract old cyclomatic number of a function from new one: $\Delta M = M(week_i) - M(week_{i-1})$
Indicator	Complexity growth: The number of functions that exceeded McCabe complexity of 20 during the last week
Model	Calculate the number of functions that exceeded cyclomatic number 20 during last week development period
Decision Criteria	If there are functions that exceeded M number 20 then software designers should review these functions refactor if necessary

5 Threats to Validity

In this paper we evaluate the validity of our results based on the framework described by Wohlin et al. [22]. The framework is recommended for empirical studies in software engineering.

The main external validity threat is the fact that our results come for an action research. However, since two companies from different domains (telecom and

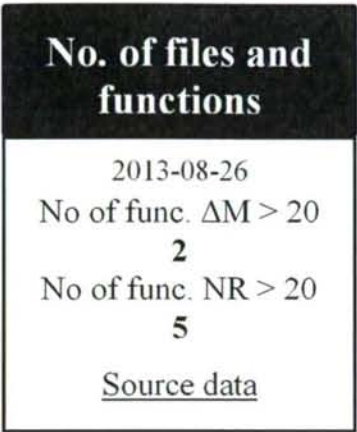


Figure 7: Information product for monitoring M and NR metrics over time

automotive) were involved, we believe that the results can be generalized to more contexts than just one specific type of software development.

The main internal validity threat is related to the construct of the study and the products. In order to minimize the risk of making mistakes in data collection we communicated the results with reference groups at both companies to validate them.

The limit 20 for cyclomatic number established as a threshold in this study does not have any firm empirical or theoretical support. It is rather an agreement of developers of large software systems. We suggest that this threshold can vary from product to product. The number 20 is a preliminary established number taking into account the number of functions that can be handled on weekly basis by developers.

The main construct validity threats are related to how we identify the names of functions for comparing their complexity numbers over time. There are several issues emerging in this operation. Namely, what happens if a function has changed its list of arguments or what happens if a function is moved to another file? Should this be regarded as the same function before and after changing the list of arguments or the position? We disregarded the change of argument list however this can be argued.

Finally the main threat to conclusion validity is the fact that we do not use inferential statistics to monitor relation between the code characteristics and project properties, e.g. number of defects. This was attempted during the study but the data in defect reports could not be mapped to individual files. This might be a thread for jeopardizing the reliability of such an analysis. Therefore we chose to rely on the most skilled designers perception of how fault-prone and unmaintainable the delivered code is.

6 Conclusions

In continuous software development quick feedbacks on developed code complexity is crucial. With small software increments there is a risk that the complexity of units of code can grow to an unmanageable level. In this paper we explored how complexity evolves, by studying two software products one telecom product at Ericsson and one automotive product at Volvo GTT. We identified that in short periods of time a few out of tens of thousands functions have significant complexity increase. We also concluded that the self-organized teams should be able to make the final assessment whether the potentially problematic is indeed problematic.

By analyzing correlations between three complexity and two change metrics we concluded that it is enough to use two measures, McCabe complexity and number of revisions, to draw attention of the teams to potentially problematic code for review and improvement.

The automated support for the teams was provided in form of a MS Sidebar gadget with the indicators and links to statistics and trends with detailed complexity development data. The measurement system was evaluated by using it on an ongoing project and communicating the results with software engineers in industry.

In our further work we intend to study how the teams formulate the decisions and monitor their implementation.

Acknowledgment

The authors thank the companies for their support in the study. This research has been carried out in the Software Centre, Chalmers, University of Gothenburg and Ericsson, Volvo Group Truck Technology.

References

- [1] Baskerville, R.L. A Critical Perspective on Action Research as a Method for Information Systems Research. *Journal of Information Technology*, 1996(11), 235-246.
- [2] Boehm, B. A view of 20th and 21st century software engineering. Paper presented at the Proceedings of the 28th international conference on Software engineering, 2006.
- [3] Bosch, Jan. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1), 67-76. doi: <http://dx.doi.org/10.1016/j.jss.2009.06.051>
- [4] Catal, Cagatay. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4), 7346-7354. doi: <http://dx.doi.org/10.1016/j.eswa.2008.10.027>

- [5] Chow, Tsun. A survey study of critical success factors in agile software projects. *Journal of Systems and Software*, 2008, 81(6), 961-971.
- [6] Fenton, Norman E. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 1999, 25(5), 675-689.
- [7] Fenton, Norman E. *Software metrics (Vol. 1)*: Chapman and Hall London, 1991.
- [8] Fitz, Timothy. Continuous Deployment at IMVU: Doing the impossible fifty times a day. from <http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/>
- [9] Garcia, F. Managing Software Process Measurement: A Meta-model Based Approach. *Information Sciences*, 2007, 177(2), 2570-2586.
- [10] Harvard Business School. *Harvard business review on measuring corporate performance*. Boston, MA: Harvard Business School Press, 1998.
- [11] International Bureau of Weights and Measures. *International vocabulary of basic and general terms in metrology = Vocabulaire international des termes fondamentaux et gneraux de mtrologie (2nd ed.)*. Genve, Switzerland: International Organization for Standardization, 1993.
- [12] Lawler, J. Measurement modeling technology. *IEEE Software*, 2003, 20(3), 68-75.
- [13] Parmenter, David. *Key performance indicators : developing, implementing, and using winning KPIs*. Hoboken, N.J.: John Wiley and Sons, 2003
- [14] Perera, G. I. U. S. Enhanced agile software development - hybrid paradigm with LEAN practice. Paper presented at the International Conference on Industrial and Information Systems (ICIIS), 2007.
- [15] Predicate Logic. TychoMetrics. Retrieved 2008-06-30, 2008, from <http://www.predicatelogic.com>
- [16] Sandberg, Anna. Agile Collaborative Research: Action Principles for Industry-Academia Collaboration. *IEEE Software*, 2011, 28(4), 74-83.
- [17] Shepperd, Martin. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2), 30-36.
- [18] Shihab, Emad. An industrial study on the risk of software changes. Paper presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012.
- [19] Developing measurement systems: an industrial case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(2), 89-107. doi: 10.1002/smr.470

- [20] Tomaszewski, Piotr. From Traditional to Streamline Development - Opportunities and Challenges. *Software Process Improvement and Practice*, 2007(1), 1-20. doi: 10.1002/spip.355
- [21] Wisell, David. Considerations when Designing and Using Virtual Instruments as Building Blocks in Flexible Measurement System Solutions. Paper presented at the IEEE Instrumentation and Measurement Technology Conference, 2007.
- [22] Wohlin, Claes. *Experimentation in Software Engineering: An Introduction*. Boston MA: Kluwer Academic Publisher, 2000.

Service Composition for End-Users

Otto Hylli*, Samuel Lahtinen*, Anna Ruokonen*, and Kari Systä*

Abstract

RESTful services are becoming a popular technology for providing and consuming cloud services. The idea of cloud computing is based on on-demand services and their agile usage. This implies that also personal service compositions and workflows should be supported. Some approaches for RESTful service compositions have been proposed. In practice, such compositions typically present mashup applications, which are composed in an ad-hoc manner. In addition, such approaches and tools are mainly targeted for programmers rather than end-users. In this paper, a user-driven approach for reusable RESTful service compositions is presented. Such compositions can be executed once or they can be configured to be executed repeatedly, for example, to get newest updates from a service once a week.

Keywords: service composition, REST, web, WADL

1 Introduction

Use of internet-based services is a routine activity for millions of users. However, the services are often silos and users do not have means to operate and manage their content across the services. Even average PC users can transfer content between applications, but nothing similar is possible for the Internet services they use. In this paper we propose an approach that allows end-users to create compositions for the purpose of combining several internet services or resources.

In service-oriented approaches dominant in the enterprise services, the focus is on the definition of service interfaces and service behavior. Service-oriented architecture (SOA) aims at loosely coupled, reusable, and composable services provided for a service consumer. SOA can be implemented by Web services, which is a technology enabling application integration. Web services can be used for composing high level composite services and business processes. Business processes are often realized as a service orchestration implemented, for example, as WS-BPEL based processes [3]. WS-BPEL is targeted for composing operation-centric Web services utilizing WSDL and SOAP [20,21]. WS-BPEL is close to a programming language defining the logic for a service orchestration. Thus, it is mostly used by IT developers.

*Department of Pervasive Computing, Tampere University of Technology, E-mail: {otto.hylli,samuel.lahtinen,anna.ruokonen,kari.systa}@tut.fi

In cloud-based systems, resources are provided to the user as services via the Internet. On the other hand, the services are accessible anywhere and through several devices. Compared to basic Internet-based service delivery, cloud adds elastic provisioning and release of computing capabilities. Cloud computing and SOA share similar interests on service reuse and service composition. Moreover cloud computing emphasizes on-demand services, which means that services should be ready for use at any time when needed. This also applies for service configurations. Thus, service configuration and composition should be enabled on-line.

Compared to business processes, typical on-demand processes for end-users are personal, simpler, and their lifetime is shorter than traditional business processes. Thus, on-demand processes are often characterized as instant service compositions and service configurations. Such processes are typically defined by the end-user instead of the developer of the cloud services. Due to instant nature of the on-demand processes, their usage and specification should be as simple as possible and require no installation of process development and management tools.

An end-user driven approach for WS-BPEL-based business process development has been proposed in [18]. The approach is targeted for providing a method for easy sketching of service orchestrations. In the proposed approach, a set of scenarios, given as UML sequence diagrams, are synthesized into a process description. However, in the context of cloud computing and on-demand processes, the use of UML modeling and standalone tools is not a proper solution.

Usually, software services in the cloud are targeted for multiple users and they provide a programmable interface, most often a Representational State Transfer (REST) API. REST is a resource-oriented architectural style developed for distributed environments such as for Web and HTTP based applications [5]. RESTful services provide an unified interface (GET, PUT, POST, DELETE) for data manipulation. Thus, composition of such services often includes combining resources and is characterized as mashup-type of development. Some guidelines for mashup development have been proposed (e.g. [14]). Thus the WB-BPEL-based approach is not applicable for cloud-based services and mashups. Composing and orchestration of RESTful services is still lacking tool vendor independent practices and description languages. Thus, the development is often done more in an ad-hoc manner.

SaaS applications are often targeted for end-users. They are self-contained and contain user-interfaces, business rules, and possibly some metadata.

A recent trend is cloud mashups, which combine resources from multiple services into a single service or application [19]. The provider of these service compositions can enhance the cloud's capabilities by offering new functionalities, which make use of existing cloud services, to clients.

In this paper, a novel approach for developing personal service compositions is presented. The approach is targeted for the end-user and allows composition of RESTful cloud services. The approach includes tackling the following issues: (1) easy sketching of service compositions using a simple visual language, (2) a mechanism to export/save composite descriptions for future usage i.e. reusable composite descriptions, and (3) an engine for executing the service compositions, once or repeatedly. The implementation of the approach called Aino service composer is currently under development. The Aino

service composer includes a web browser based editor, which can be used to create simple on-demand service compositions. An earlier version of the tool description has been published in [9].

The rest of the paper is organized as follows. In Section 2, we describe the overall approach and related components. In Section 3, two use cases for end-user driven service composition are presented. Aino service composer is described in Section 4. In Section 5, related work and topics are discussed. In Section 6, conclusions and plan for future work are presented.

2 User-driven approach for service composition

In this paper, an end-user driven approach for defining personal service compositions is presented. The main goal of the approach is on easy design of service compositions, which requires minimal technical knowledge. The service composition is created by using GUI widgets, which are generated based on an imported service description. Widgets present individual resources and they can be dragged and dropped on the canvas. The user can draw dataflow pipes to connect the widgets. Incoming and outgoing dataflows are mapped to REST methods (e.g. outgoing dataflow for GETting a resource presentation).

The implementation of the approach called Aino service composer consists of two components, designer Ilmarinen and engine Sampo. Ilmarinen is a client side application for creating and editing compositions and it is run in a web browser. Sampo is a server side application, which is an engine for running the service compositions. The composition description is given in XML-based format, called Aino description. As a service description format, the approach is based on WADL descriptions [22]. It defines the resources, i.e., URIs, methods, and parameters. That is, while the Aino description specifies the service logic, the WADL description describes the service interface.

Sampo also plays a role of a service registry. Once a service is registered in Sampo, it can be used as a constituent service for future applications. One reason for providing a centralized registry, instead of letting the user search from the web, is that for RESTful services there is no agreement on one service description format. In case a third-party service does not have a compatible WADL description, it can be created afterwards and registered to Sampo. Thus, the approach allows using services, which do not natively provide a WADL description, as reusable constituents.

The approach includes the following steps:

- (1) query services from the service registry,
- (2) select services to be used as a part of the composition,
- (3) composition described as a data flow between services, and
- (4) send the composition description to the server engine to be executed.

The main steps are shown in Fig. 1. It also shows the relations between the main components of the system and descriptions, Aino and WADL, which are used for importing and exporting data (i.e. service and composition descriptions).

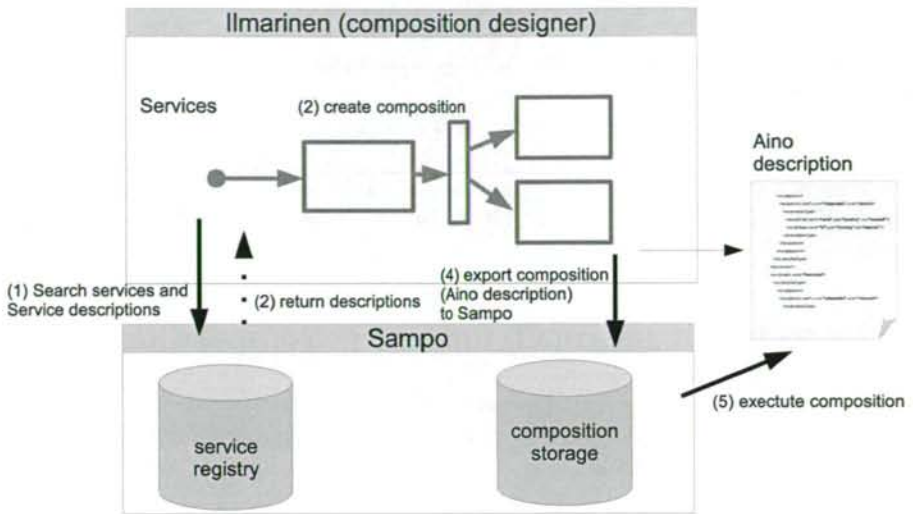


Figure 1: The main steps of the approach

3 Use cases

The following two use cases illustrate the possibilities offered by service compositions for regular internet users. They show how after encountering a normally labor intensive internet based task including multiple services, a user can pretty easily create a service composition that takes care of the task.

3.1 Use case 1: Photos from Twitter to Flickr selectively

An avid Twitter¹ user has been sending many photos taken with his smart phone directly to Twitter. The user wants a better way to organize and share his photos so he opens an account in Flickr² which enables him to save photos to different albums, associate keywords to them and decide which photos are public. Uploading all his photos manually to Flickr would be tedious for the user. He would have to go through his Twitter time line, download each photo to his computer and then upload it to Flickr.

To automate the upload process the user wants to create a service composition with Aino service composer. He opens the composition designer Ilmarinen and chooses that he wants to get photos. Ilmarinen shows him a list of services from where he can get photos and he chooses Twitter. From Twitter he chooses that he wants photos from one user which in this case is himself. He also indicates that all photos shouldn't be fetched, instead he will select the ones he wants. Then the user tells Ilmarinen that he wants to upload the photos selected in the previous step. From the services list shown by Ilmarinen he chooses Flickr as the upload target. Additionally he specifies that he wants to choose for each photo

¹ www.twitter.com

² www.flickr.com

whether it is private or public. Lastly, he tells Ilmarinen that he wants to delete photos and chooses Twitter. He specifies that from Twitter he wants to delete those photos he has marked as private for Flickr.

When he executes the composition the execution engine Sampo first asks him to authorize Sampo's use of his Twitter and Flickr accounts. Authorization will be done by using OAuth [10] which means that the user authenticates to both services which then give access tokens to Sampo. Sampo will store these access tokens for later use if the user wants it so that next time a service composition using these services is run the user doesn't need to authenticate to the services. He just has to log in to Sampo. When the actual execution has started Sampo will first show the user all his photos from Twitter and asks him to choose those he wants. After that Sampo shows the user his previously chosen photos and asks which of them he wants to be private in Flickr. After the execution has finished Sampo shows the user a execution results summary which tells that the execution was a success and shows how many photos were processed in each step.

3.2 Use case 2: Affordable reading

An enthusiastic book reader uses the Goodreads³ service to support her hobby. Goodreads is an online community for readers where users can search for books, rate and review them. Users can also categorize books in their profile by adding them to different shelves. One of these shelves is to-read where the user has been adding interesting books, which she has found through Goodreads' recommendation system. She wants to buy some new reading from her to-read shelf but due to her current poor economic situation she wants it to be as cheap as possible. Searching for each book's price from her favorite online book retailer Amazon⁴ and then comparing the prices manually would be time consuming so she decides to create a service composition to make the process quicker.

The user opens the service composition designer Ilmarinen and chooses that she wants information about books. Ilmarinen gives the user a list of services that deal with books. The user chooses Goodreads and indicates that she wants the content of a particular user's, in this case hers, particular shelf. Ilmarinen asks the user to input the name of the user and the name of the shelf which in this case are the user's Goodreads user name and to-read. Next the user tells Ilmarinen that she wants online shopping services. From the service list she chooses amazon.com. She specifies that she wants product information about the books from the previous step. Lastly she tells Ilmarinen that she wants the results in ascending order by price. When this composition is run the result is a table containing book information from Amazon including the price and a link to the Amazon product page where the book can be bought.

4 Implementation

The prototype implementation of the Aino service composer consists of two main components: Designer Ilmarinen and engine Sampo. Sampo executes the service compositions,

³www.goodreads.com

⁴www.amazon.com

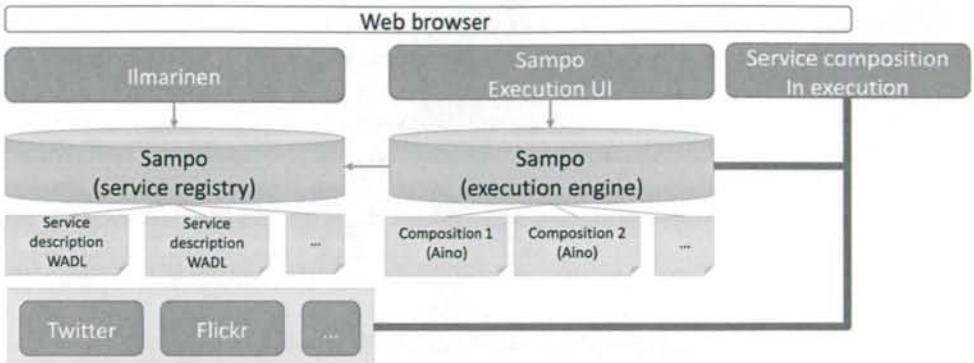


Figure 2: High level architecture of the Aino service composer

stores the service descriptions and offers Ilmarinen access to the information. The separation of the two main components allows their more independent development. Figure 2 illustrates the high-level architecture of the Aino service composer. The user uses browser-based Ilmarinen to create service compositions. A service composition is a service. Its interface is defined as a WADL document and its execution instructions are defined with the Aino composition description language. Both XML documents are stored in Sampo. The user interacts with engine component Sampo which is used to execute the compositions. The execution and possible user interaction related to the execution is again done in a browser based UI.

4.1 Service description

All the constituent services, as well as the service composition, are described with WADL documents. WADL description defines the service, provided methods and their parameters, as well as data types. The data types can also be defined as separate XML schema files. An example of a simple service description is shown below. It has a partial definition of Twitter's get user timeline method which returns a specified number of tweets from the given user.

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
  <grammars></grammars>
  <resources base="https://api.twitter.com/1.1">
    <resource path="statuses/user_timeline.json">
      <method href="getTimeline"/>
    </resource>
  </resources>
  <method name="GET" id="getTimeline">
    <request>
      <param name="screen_name" style="query" type="xsd:string" />
      <param name="count" style="query" type="xsd:integer" />
    </request>
    <response>
      <representation mediaType="application/json" />
    </response>
  </method>
</application>
```

```
</method>  
</application>
```

4.2 Engine Sampo

Engine Sampo is used in two ways, as a service registry and as an engine to execute the service compositions. Services can be added in the service registry as WADL descriptions. It provides the basic functionality for registration of the services, i.e. API for adding, removing, and searching the services. When a new WADL is added to Sampo the part of the categorization of the service and the resources can be done automatically based on the WADL and an expert user, who understands rest services and WADL, can complete the information and extend the suggested categorizations.

The given metadata is used to offer Ilmarinen lists of the services. For instance, the user can ask to get a list of services related to pictures. Thanks to the metadata Ilmarinen only needs to process WADLs of the services user adds to her composition instead of processing every WADL.

The other part of Sampo provides a REST interface for adding and executing Aino descriptions. The service composition execution uses Aino and the corresponding WADL descriptions for getting the required information on the services and their API. The engine uses this information to invoke correct API calls to the services and combine the tasks to create the complete composite service.

Sampo contains a user interface for handling the compositions. The user can parameterize the composition and define time intervals of execution. In case of a recurring task the service page can be used to start and stop the compositions and change their time intervals. For instance, one could define a service composition that is launched weekly.

Sampo implements simple basic services, for example, for displaying images and news feeds. These are available as components in Ilmarinen and can be added to a service composition in similar fashion as external services.

Sampo is implemented as a Java based web application with the Spring framework⁵. Sampo's implementation is ongoing work. Features that require work include making Sampo work with a greater number of data types and implementing metadata editing for services.

4.3 Designer Ilmarinen

Ilmarinen is a client side application, which provides a graphical interface for creating the service compositions. The user is provided a simple visual environment for defining the service composition. The composition is done partially in a guided manner. A screenshot of an early prototype version of the tool is shown in Figure 3. The user can choose the services e.g. Twitter, BBC Program guide, Weather) she wants based on the service category (e.g. Social media, file storage, picture, program guides). For the services the user can define the interaction and the resources related to the interaction.

In the service composition key elements are the services and data flow between them. After adding a service one can see the input and output possibilities offered by it. These

⁵<http://projects.spring.io/spring-framework/>

inputs and outputs are parameterized and services are connected to each other using them. When the user has finished, Ilmarinen generates the Aino description. This is exported to Sampo engine for execution. The composition is stored in Sampo and can be accessed directly using a corresponding link. That allows the users to access and execute the compositions directly without using Ilmarinen. This also enables sharing service compositions among different users.

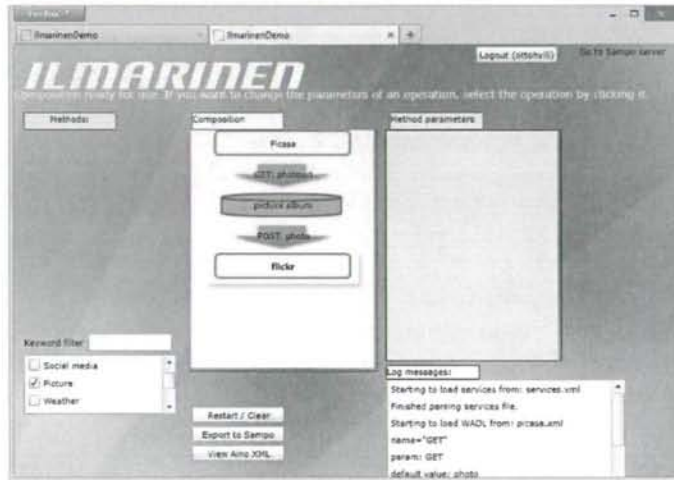


Figure 3: Screenshot of Prototype of Ilmarinen

4.4 Composite description language Aino

Descriptions written in Aino language define the services and resources involved in the composition and the dataflow. A dataflow from one service to another means by getting resource presentation from one service with GET methods and using it as an input to another service using PUT, POST, or GET methods. Services can provide three types of resources: resource out (for GETting a representation), resource in (for PUTting or POSTing), and resource in/out (for PUTting or POSTing and GETting). For data manipulation, control nodes, such as merge and select nodes, are used.

The dataflow can be modeled as an acyclic graph structure, which consists of resources, control nodes, and dataflow connections between them. Control nodes are used for manipulating resource representations, e.g. transforming or filtering data.

In addition to resource, control nodes and dataflow connections, the dataflow includes definition of method calls that are executed when the composition is run. These method calls to the services are presented as GET, PUT, POST, and DELETE elements in the XML description. In addition, the composite service can receive method calls from other compositions using this as a service or from user agent initiation. These are presented as onPUT, onGET, onPOST, and onDELETE elements. Corresponding request and response message types (including data types) are described in the services' WADL documents.

These activities corresponding to REST operations are the same, which are used in BPEL for REST [16] proposal.

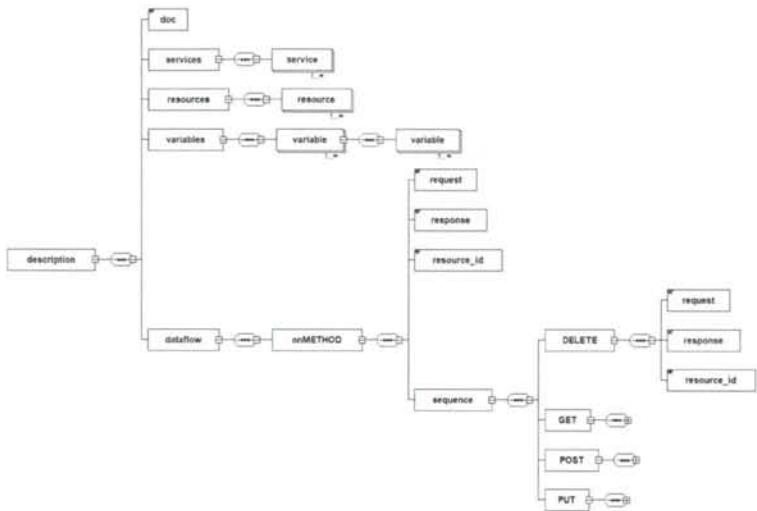


Figure 4: Aino language structure

To enable importing and exporting of compositions, Aino descriptions are transformed in XML format. The structure of Aino language is given in Figure 4. It is explained in detail using an example Aino description given below. The given description presents an example of sending links from Twitter tweets to Instapaper⁶. Instapaper is a service where users can add links to articles they found from the web that they want to read later. Resources part defines two resources, Twitter’s user timeline and instapaper’s add, which participate in the composition. User timeline returns the desired number of tweets from the specified user. Its WADL was an example in section 4.1. Instapaper’s add resource adds the link in the url parameter to the account whose username and password are in the respective parameters.

The example composition consists of a receive message and two message invocations. Execution starts when the client invokes GET method on the composite resource (onGET element). Execution continues with a sequence of two invocations. First the composite service invokes GET method on Twitter and second it invokes POST method on Instapaper.

```
<?xml version="1.0" encoding="UTF-8"?>
<description name="tweetlinks2instapaper" >
<doc>Send links from the 10 most recent tweets from the specified user to Instapaper.</doc>
<services>
  <service name = "twitter" id="52d" />
  <service name = "instapaper" id="52f" />
</services>
```

⁶www.instapaper.com

```

<resources>
  <resource uri="https://api.twitter.com/1.1/statuses/user_timeline.json"
    resource_id="r1" service_id="52d" />
  <resource uri="https://www.instagram.com/api/add" resource_id="r2" service_id="52f" />
</resources>

<variables>
  <variable name="twitterparams" type="variablesset" >
    <variable name="screen_name" type="string" open="true" />
    <variable name="count" type="integer" value="10" />
  </variable>
  <variable name="links" type="linklist" />
  <variable name="instapaperparams" type="variablesset" >
    <variable name="username" type="string" value="john.smith@gmail.com" />
    <variable name="password" type="string" value="password123" />
    <variable name="urls" type="variablereference" value="links" />
  </variable>
</variables>

<dataflow>
  <onGET>
    <request></request>
    <response>links</response>
    <resource_id>r_comp</resource_id>
    <sequence>
      <GET>
        <request>twitterparams</request>
        <response>links</response>
        <resource_id>r1</resource_id>
      </GET>
      <POST>
        <request>instapaperparams</request>
        <response></response>
        <resource_id>r2</resource_id>
      </POST>
    </sequence>
  </onGET>
</dataflow>
</description>

```

Variables are used for storing and manipulating message values. For example, the given code listing defines three variables, which correspond to input and output message types of the used GET and POST methods. The variables *twitterparams* and *instapaperparams* are of the type *variablesset* which means that they contain multiple variables. These variables contain the parameters for the requests to the services. This is indicated in the Aino description by putting them into the request elements of the service call. The member variables of these sets *screen_name*, *count*, *username* and *password* correspond directly to parameters defined in services' WADLs. So for example Twitter's user timeline method has a parameter named *screen_name*. The variable *links* contains a list of links. Links from the Twitter method call's response are saved to this variable. How this information is extracted from the response is explained in section 4.5. The *links* variable is also the response of the composition which means that it will be shown to the user. The variable is also one of the request parameters for Instapaper because of the variable reference in the *instapaperparams*. Because Instapaper's api doesn't support sending multiple links in one request, the execution engine has to make multiple post requests but this detail doesn't matter to the Aino description.

screen_name is initialized, when the user fills in the required input data, when she

decides to run the composition (see Figure 5). A control interface is used for specifying process instance specific information, such as initial value of process variables and repetition information, which is not part of Aino description.

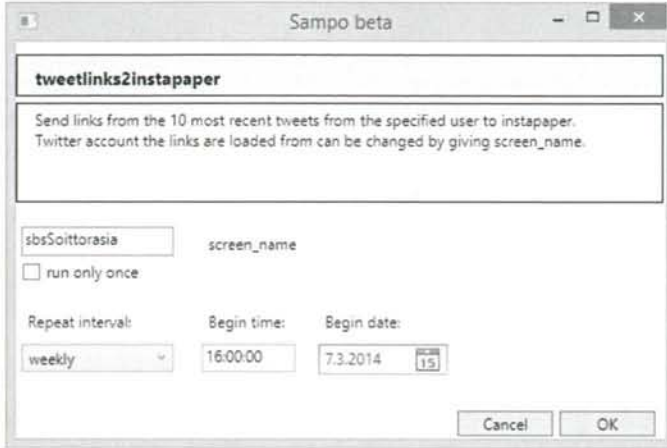


Figure 5: A Control User Interface for the service Compositions

4.5 Data processing

One challenge in combining different internet services into compositions are the different ways the services represent the same data. Many services deal with the same kind of data, e.g. photos or status updates. However, these services represent this data in different ways. One uses XML in representing its resources while another uses JSON. Even if both services in a composition use the same format the schema would very probably be different. Below is an example of how Twitter and Facebook represent a status update. Both service's status update contains the name of the poster, time of the posting and the actual content of the status. They have different names for these attributes and they also have a different time format for the posting time. In addition, each service has additional service specific information about the status update which is not shown here.

Facebook:

```
{
  "id": "201192066592832",
  "from": {
    "name": "Otto Hylli",
    "id": "10883825396030"
  },
  "message": "Hello, world.",
  "updated_time": "2012-05-15T20:35:25+0000",
}
```

Twitter:

```
{
  "text": "Hello, world",
  "id": 377326766385573888,
```



```

"user": {
  "id": 918830997,
  "name": "Otto Hylli"
},
"created_at": "Mon Aug 16 17:45:23 +0000 2013"
}

```

Our solution for this problem is to define a set of generic data types that internet services provide and consume. These types include among others status update, photo, link, location and product. For each data type we define a group of attributes that this kind of data generally has. For instance, a status update has the name of the poster, the content of the status update, and the time of the posting. For a service that returns representations that correspond to a certain data type the representation needs to be mapped to the data type. For example, in Twitter's and Facebook's cases mapping information tells how to build a status object from the JSON. For instance, Where in JSON the posting time of the update is and what the format of the time information is. This means that we need mechanisms to locate the interesting data from a structured document.

For locating the desired information from the representation we use XPath [23] for XML representations and JsonPath [6] for JSON. XPath is a language for addressing specific parts of a XML document. It is based on XPath expressions which select the specified nodes from the XML. JsonPath is a similar system for JSON. XPath and JsonPath based data processing information can be added by an expert user directly to a service's WADL or to the service's metadata in the service registry. In both cases the metadata will contain the required formatting information such as the time format used. For instance, Twitter's time format can be represented with this pattern string: E MMM d H:m:s Z y. The pattern format used is from the standard Java class used in the implementation to parse dates.

In the WADL XPath or JsonPath information is located inside the representation element of a resource's method's response. The information itself is contained in the param elements. The parameter's name is a keyword that tells what kind of information it contains, e.g. the author of a status update. The path attribute of the parameter contains the XPath or JSONPath expression itself. The example below shows the representation elements for Twitter's and Facebook's methods that return a list of status updates. A more refined description of the generic data types and service metadata that uses them will be published in [8].

Twitter:

```

<representation mediaType="application/json">
  <param name="status_text" type="xsd:string" path="$[*].text" />
  <param name="status_creator" type="xsd:string" path="$[*].user.name" />
  <param name="status_posted" type="xsd:string" path="$[*].created_at" />
</representation>

```

Facebook:

```

<representation mediaType="application/json">
  <param name="status_text" type="xsd:string" path="$[*].message" />
  <param name="status_creator" type="xsd:string" path="$[*].from.name" />
  <param name="status_posted" type="xsd:string" path="$[*].updated_time" />
</representation>

```

5 Related work

The idea of cloud computing is based on on-demand services, which are provided as SaaS applications. In the cloud, traditional business process management tools are already available as SaaS. However, they are targeted for design and management of structured business processes. Requirements for on-demand processes differ from traditional BPM. The ideal solution is to provide an easy and instant mechanism to support execution of personal and dynamic processes, which utilize existing SaaS applications available on the cloud.

5.1 Tools for mashup development

Ad-hoc processes are often expected to live only for a short time. The lack of documentation and proper design might make them single-use only. Thus, they may not be reusable and flexible, but they always need to be recomposed.

JOpera [15] is an Eclipse-based tool build for composing SOAP/WSDL and RESTful Web services. For software developers it provides many useful features such as process modeling, debugging and execution. For composing RESTful services JOpera uses BPEL for REST [16]. BPEL for REST is an extension to WS-BPEL to support compositions of RESTful Web services. The approach does not rely on usage of WSDL or other service descriptions. Resources are defined in the BPEL for REST description as a resource construct, which defines the resource URI and supported operations.

In [13], Marino *et al.* present HTML5-based prototype tool support for mashup development. They present a visual language for service composition. However, the paper is missing details on the user interface and tool usage. Also, details on the composition description are not given.

In [1], Aghee *et al.* discuss different types of mashups enabled by HTML5. A case example includes a location sensitive mobile mashup. The mashup runs natively in a mobile device and uses the GPS sensor build-in the device. In addition, it uses external Web APIs. Location data is sent to a server, which executes API calls to external services. This enables sharing the application between several uses. Mobile mashups enable use of real-time data gathered from the sensors in a mobile phone, e.g. real-time navigation.

Bottaro *et al.* present a simple visual language for composing location-based services [4]. The user uses a repository of web widgets. Widgets are dragged and dropped to build UI for the application. The application logic is defined by drawing connections between data widgets.

In [7], Grönvall *et al.* present ongoing work on user-centric service composition. GUI elements are prototypes of service invocations, which can be chained to compose data flows among services. They present a lightweight tool support for composing simple dynamic workflows, such as for combining SMS, email, and calendar services. Instead of modeling complicated workflows, the emphasis is on the user experience.

In EzWeb project [11, 12], a service-oriented platform for end-user mashup development has been built. The idea is to provide gadgets (e.g. Twitter, Flickr) the user could add to her "application page" creating a set of different applications and web services. The user can also define dataflow between the gadgets by connecting "events" the gadgets could give, e.g., an image url could be connected to another image displayer gadget that is

able to show the picture. All these gadgets are implemented for EzWeb environment. That is, implementation of their user interface, the way of communicating with servers, their events and event slots, are specific for the EzWeb environment. In our approach, the aim is to provide means to compose existing services together and execute these compositions. Thus, our target is to support composition of any third party services by introducing their service descriptions to our system.

5.2 Describing service compositions

Some approaches for modeling and describing RESTful service compositions have been proposed. Guidelines for UML modeling of RESTful service compositions is presented in [17] by Rauf *et al.* The static resource structure is modeled using class diagrams. The behavioral specification of the composite service is given using state chart diagrams.

In [24, 25], Zhao *et al.* discuss formal describing of RESTful services and resources as well as RESTful composite services. Their main interests is on supporting automatic service compositions. For service compositions they present a logic-based synthesis approach utilizing linear-logic and π -calculus.

In [2], Alarcon *et al.* state that many of the recent service composition approaches rely on operation-based models and neglect hypermedia characteristics of REST. As a solution for composing RESTful services, they present a hypermedia-driven approach realized by using resource linking language (ReLL) for service description. The approach aims to support machine-clients by enabling automatic retrieving of resources from a web site. For describing the composite resources PetriNets are used. As an example of a composite resource, a social network application was presented.

6 Conclusions

Cloud computing is based on on-demand services, which should be available as needed. Similarly, it should also enable on-demand service compositions. In this paper, an end-user driven approach for personal service composition has been presented. The proposed tool support i.e. Aino service composer includes a composition designer running in a web browser and a server-side engine for storing and executing service compositions. The designer is designed for the end-users and it is used for creating personal service compositions. It focuses on end-user concepts and aims to hide complicated and unnecessary information, e.g. service descriptions, which are handled by the engine. Instead of handling data types, the user is allowed to use concepts such as a picture or a photo gallery. The presented use cases concentrate on combining social media services into a composite service. Also, the user is allowed to define repeatable executions for checking updates from the services.

To characterize the approach, it is designed for cloud environment providing a browser-based tool for building service compositions. It is based on WADL descriptions, which are also used for generating GUI widgets for the end-user. In addition, it enables defining RESTful workflows as a composite service.

Our future work includes finalizing the implementation and conducting case studies on applying the approach utilizing the developed tool support. Our future plans also include experimenting the tool usage with novice users.

References

- [1] Aghaee, S. and Pautasso, C. Mashup development with HTML5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.
- [2] Alarcon, R., Wilde, E., and Bellido, J. Hypermedia-driven RESTful service composition. In *Proceedings of the 2010 international conference on Service-oriented computing*, ICSOC'10, pages 111–120, Berlin, Heidelberg, 2011. Springer-Verlag.
- [3] Andrews, T., Curbera, F., Dholakia, H., Golan, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. Business Process Execution Language for Web Services Version 1.1, May 2003. <http://www.ibm.com/developerworks/>.
- [4] Bottaro, A., Marino, E., Milicchio, F., Paoluzzi, A., Rosina, M., and Spini, F. Visual programming of location-based services. In *Proceedings of the 2011 international conference on Human interface and the management of information - Volume Part I*, HI'11, pages 3–12, Berlin, Heidelberg, 2011. Springer-Verlag.
- [5] Fielding, R.T. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
- [6] Goessner, S. Jsonpath - xpath for json. <http://goessner.net/articles/JsonPath/>.
- [7] Grönvall, E., Ingstrup, M., Pløger, M., and Rasmussen, M. Rest based service composition: Exemplified in a care network scenario. In Costagliola, G., Ko, A.J., Cypher, A., Nichols, J., Scaffidi, C., Kelleher, C., and Myers, B.A., editors, *VL/HCC*, pages 251–252. IEEE, 2011.
- [8] Hylli, O., Lahtinen, S., Ruokonen, A., and Systä, K. Resource description for end-user driven service compositions. Submitted to 2nd International Workshop on Personalized Web Tasking (PWT 2014), 2014.
- [9] Hylli, O., Lahtinen, S., Ruokonen, A., and Systä, K. Service composition for end-users. In *13th Symposium on Programming Languages and Software Tools (SPLST'13)*, page pp.15, 2013.
- [10] Internet Engineering Task Force (IETF), <http://tools.ietf.org/html/rfc6749>. *The OAuth 2.0 Authorization Framework*, 2012.
- [11] Lizcano, D., Soriano, J., Reyes, M., and Hierro, J.J. EzWeb/FAST: Reporting on a successful mashup-based solution for developing and deploying composite applications in the "upcoming ubiquitous SOA". In *Mobile Ubiquitous Computing, Systems*,

Services and Technologies, 2008. UBIComm '08. The Second International Conference on, pages 488–495, 2008.

- [12] Lizcano, D., Soriano, J., Reyes, M., and Hierro, J.J. EzWeb/FAST: reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, ii-WAS '08*, pages 15–24, New York, NY, USA, 2008. ACM.
- [13] Marino, E., Spini, F., Minuti, F., Rosina, M., Bottaro, A., and Paoluzzi, A. HTML5 visual composition of rest-like web services. In *4th IEEE International Conference on Software Engineering and Service Science (ICSESS 2013)*, 2013. To appear.
- [14] Mikkonen, T. and Salminen, A. Towards a reference architecture for mashups. In *Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems, OTM'11*, pages 647–656, Berlin, Heidelberg, 2011. Springer-Verlag.
- [15] Pautasso, C. Composing RESTful services with JOpera. In *International Conference on Software Composition 2009*, volume 5634, pages 142–159, Zurich, Switzerland, July 2009. Springer.
- [16] Pautasso, C. RESTful web service composition with BPEL for REST. *Data Knowl. Eng.*, 68(9):851–866, September 2009.
- [17] Rauf, I., Ruokonen, A., Systä, T., and Porres, I. Modeling a composite RESTful web service with UML. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA '10*, pages 253–260, New York, NY, USA, 2010. ACM.
- [18] Ruokonen, A., Pajunen, L., and Systä, T. Scenario-driven approach for business process modeling. *Web Services, IEEE International Conference on*, 0:123–130, 2009.
- [19] Singhal, M., Chandrasekhar, S., Ge, T., Sandhu, R., Krishnan, R., Ahn, G-J., and Bertino, E. Collaboration in multicloud computing environments: Framework and security issues. *Computer*, 46(2):76–84, 2013.
- [20] W3C, <http://www.w3.org/TR/wsdl>. *Web Services Description Language (WSDL) 1.1*, 2001.
- [21] W3C, <http://www.w3.org/>. *Simple Object Access Protocol (SOAP) 1.2*, 2007. Last visited December 2011.
- [22] W3C, <http://www.w3.org/Submission/wadl/>. *Web Application Description Language (WADL)*, 2009.
- [23] W3C, <http://www.w3.org/>. *XML Path Language (XPath) 2.0 (Second Edition)*, 2010.

- [24] Zhao, H. and Doshi, P. Towards automated RESTful web service composition. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 189–196, July.
- [25] Zhao, X., Liu, E., Clapworthy, G.J., Ye, N., and Lu, Y. RESTful web service composition: Extracting a process model from linear logic theorem proving. In *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, pages 398–403, Oct.

Extensions to the CEGAR Approach on Petri Nets*

Ákos Hajdu[†], András Vörös[‡], Tamás Bartha[‡] and Zoltán Mártonka[†]

Abstract

Formal verification is becoming more prevalent and often compulsory in the safety-critical system and software development processes. Reachability analysis can provide information about safety and invariant properties of the developed system. However, checking the reachability is a computationally hard problem, especially in the case of asynchronous or infinite state systems. Petri nets are widely used for the modeling and verification of such systems. In this paper we examine a recently published approach for the reachability checking of Petri net markings. We give proofs concerning the completeness and the correctness properties of the algorithm, and we introduce algorithmic improvements. We also extend the algorithm to handle new classes of problems: submarking coverability and reachability of Petri nets with inhibitor arcs.

Keywords: Petri Nets, reachability analysis, abstraction, CEGAR

1 Introduction

The development of complex, distributed systems, and safety-critical systems in particular, requires mathematically precise verification techniques in order to prove the suitability and faultlessness of the design. Formal modeling and analysis methods provide such tools. However, one of the major drawbacks of formal methods is their computation and memory-intensive nature: even for relatively simple distributed, asynchronous systems the state space and the set of possible behaviors can become unmanageably large and complex, or even infinite.

This problem also appears in one of the most popular modeling formalisms, Petri nets. Petri nets have a simple structure, which makes it possible to use strong structural analysis techniques based on the so-called *state equation*. As structural analysis is independent of the initial state, it can handle even infinite

*This work was partially supported by the European Union and the European Social Fund through the project FuturICT.hu (grant no. TAMOP-4.2.2.C-11/1/KONV-2012-0013) of VIKING Zrt Balatonfured.

[†]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary. E-mail: vori@mit.bme.hu

[‡]Institute for Computer Science and Control, MTA SZTAKI, Budapest, Hungary.

state problems. Unfortunately, its pertinence to practical problems, such as reachability analysis, has been limited. Recently, a new algorithm [13] using Counter-Example Guided Abstraction Refinement (CEGAR) extended the applicability of state equation based reachability analysis.

Our paper improves this new algorithm in several important ways. The authors of the original CEGAR algorithm have not published proofs for the completeness of their algorithm and the correctness of a heuristic used in the algorithm. In this paper we analyze the correctness and completeness of their work as well as our extensions. We prove the lack of correctness in certain situations by a counterexample, and provide corrections to overcome this problem. We also prove that the algorithm is incomplete due to its iteration strategy. We describe algorithmic improvements that extend the set of decidable problems, and that effectively reduce the search space. We extend the applicability of the approach even further: we provide solutions to handle Petri nets with inhibitor arcs, and the so-called *submarking coverability* problem. At the end of our paper we demonstrate the efficiency of our improvements by measurements.

2 Background

In this section we introduce the background of our work. First, we present Petri nets (Section 2.1) as the modeling formalism used in our work. Then we introduce the counterexample guided abstraction refinement method and its application for the Petri net reachability problem (Section 2.2).

2.1 Petri nets

Petri nets are graphical models for concurrent and asynchronous systems, providing both structural and dynamical analysis. Formally, a Petri net is a tuple $PN = (P, T, E, W)$, where P is the set of *places*, T is the set of *transitions*, with $P \neq \emptyset \neq T$ and $P \cap T = \emptyset$, $E \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs* and $W : E \rightarrow \mathbb{Z}^+$ is the weight function assigning weights $w^-(p_j, t_i)$ to the edge $(p_j, t_i) \in E$ and $w^+(p_j, t_i)$ to the edge $(t_i, p_j) \in E$ [9].

A *marking* of a Petri net is a mapping $m : P \rightarrow \mathbb{Z}_0^+$. A place p contains k tokens under a marking m if $m(p) = k$. The initial marking is usually denoted by m_0 .

A transition $t_i \in T$ is *enabled* in a marking m , if $m(p_j) \geq w^-(p_j, t_i)$ holds for each $p_j \in P$ with $(p_j, t_i) \in E$. An enabled transition t_i can *fire*, consuming $w^-(p_j, t_i)$ tokens from places $p_j \in P$ with $(p_j, t_i) \in E$ and producing $w^+(p_j, t_i)$ tokens in places $p_j \in P$ with $(t_i, p_j) \in E$. The firing of a transition t_i in a marking m is denoted by $m[t_i]m'$ where m' is the marking after firing t_i .

A word $\sigma \in T^*$ is a *firing sequence*. A firing sequence is *realizable* in a marking m and leads to m' , (denoted by $m[\sigma]m'$), if either $m = m'$ and σ is an empty word, or there exists a realizable firing sequence $w \in T^*$, a transition $t_i \in T$, and a marking m'' such that $m[w]m''[t_i]m'$. The *Parikh image* of a firing sequence σ is a vector $\varphi(\sigma) : T \rightarrow \mathbb{Z}_0^+$, where $\varphi(\sigma)(t_i)$ is the number of the occurrences of t_i in σ .

Petri nets can be extended with *inhibitor arcs* to become a tuple $PN_I = (PN, I)$, where $I \subseteq (P \times T)$ is the set of inhibitor arcs. There is an extra condition for a transition $t_i \in T$ with inhibitor arcs to be enabled: for each $p_j \in P$, if $(p_j, t_i) \in I$, then $m(p_j) = 0$ must hold. Petri nets extended with inhibitor arcs are *Turing complete* [10].

Reachability problem. A marking m' is *reachable* from m if there exists a realizable firing sequence $\sigma \in T^*$, for which $m[\sigma]m'$ holds. The set of all reachable markings from the initial marking m_0 of a Petri net PN is denoted by $R(PN, m_0)$. The aim of the *reachability problem* is to check if $m' \in R(PN, m_0)$ holds for a given marking m' .

We define a *predicate* as a linear inequality on markings of the form $Am \geq b$, where A is a matrix and b is a vector of coefficients [6]. The aim of the *submarking coverability problem* is to find a reachable marking $m' \in R(PN, m_0)$, for which the given predicate $Am' \geq b$ holds.

The reachability problem is decidable [8], but it is at least EXPSPACE-hard [7]. Using inhibitor arcs, the reachability problem in general is undecidable [3].

State equation. The *incidence matrix* of a Petri net is a matrix $C_{|P| \times |T|}$, where $C(i, j) = w^+(p_i, t_j) - w^-(p_i, t_j)$. Let m and m' be markings of the Petri net, then the *state equation* takes the form $m + Cx = m'$. Any vector $x \in (\mathbb{Z}_0^+)^{|T|}$ fulfilling the state equation is called a *solution*. Note that for any realizable firing sequence σ leading from m to m' , the Parikh image of the firing sequence fulfills the equation $m + C\wp(\sigma) = m'$. On the other hand, not all solutions of the state equation are Parikh images of a realizable firing sequence. Therefore, the existence of a solution for the state equation is a necessary but not sufficient criterion for the reachability. A solution x is called *realizable* if a realizable firing sequence σ exists with $\wp(\sigma) = x$.

T-invariants. A vector $x \in (\mathbb{Z}_0^+)^{|T|}$ is called a *T-invariant* if $Cx = 0$ holds. A realizable T-invariant represents the possibility of a cyclic behavior in the modeled system, since its complete occurrence does not change the marking. However, during firing the transitions of the invariant, some intermediate markings can be interesting for us.

Solution space. Each solution x of the state equation $m + Cx = m'$, can be written as the sum of a *base vector* and the linear combination of T-invariants [13], which can formally be written as $x = b + \sum_i n_i y_i$, where b is a base vector and n_i is the coefficient of the T-invariant y_i .

2.2 The CEGAR approach

Counterexample guided abstraction refinement (CEGAR) is a general approach for analyzing systems with large or infinite state spaces. The CEGAR method works

on an abstraction of the original model, which has a less detailed state space representation. During the iteration steps, the CEGAR method refines the abstraction using the information from the explored part of the state space. When applying CEGAR on the Petri net reachability problem [13], the initial abstraction is the state equation. Solving the state equation is an integer linear programming problem [5], for which the ILP solver tool can yield one solution, minimizing a target function of the variables. Since the algorithm seeks the shortest firing sequences leading to the target marking, it minimizes the function $f(x) = \sum_{t \in T} x(t)$. The feasibility of the state equation is a necessary, but not sufficient criterion for reachability, so the following situations are possible:

- If the state equation is infeasible, the necessary criterion does not hold, thus the target marking is not reachable.
- If the state equation has a solution which is realizable by some firing sequence, the target marking is reachable.
- If the state equation has an unrealizable solution, it is a counterexample and the abstraction has to be refined.

The purpose of the abstraction refinement is to exclude counterexamples from the solution space without losing any realizable solutions. For this purpose, the CEGAR approach uses linear inequalities over transitions, called *constraints*.

Constraints. Two types of constraints were defined by Wimmel and Wolf [13]:

- *Jump constraints* have the form $|t_i| < n$, where $n \in \mathbb{Z}_0^+$, $t_i \in T$ and $|t_i|$ represents the firing count of the transition t_i . Jump constraints can be used to switch between base vectors, exploiting their pairwise incomparability.
- *Increment constraints* have the form $\sum n_i |t_i| \geq n$, where $n_i \in \mathbb{Z}$, $n \in \mathbb{Z}_0^+$, and $t_i \in T$. Increment constraints can be used to reach non-base solutions.

As an example, consider the Petri net in Figure 1(a) with the reachability problem $(1, 0, 1, 0) \in R(PN, (0, 0, 1, 0))$. There are two base vectors for this problem: $(1, 0, 0)$ (firing t_0) and $(0, 1, 1)$ (firing t_1 and t_2). The ILP solver yields the solution $(1, 0, 0)$ first, which is unrealizable, but using the jump constraint $|t_0| < 1$, the ILP solver can be forced to produce the realizable solution $(0, 1, 1)$. Consider now the Petri net in Figure 1(b) with the reachability problem $(1, 0, 1) \in R(PN, (0, 0, 1))$. The only base vector for this problem is the vector $(1, 0, 0)$ (firing t_0), which is unrealizable. Using an increment constraint $|t_1| \geq 1$, the ILP solver can be forced to add the T-invariant $\{t_1, t_2\}$ to the new solution $(1, 1, 1)$, which is realizable by the firing sequence $\sigma = (t_1, t_0, t_2)$.

2.2.1 Partial solutions

Given a Petri net $PN = (P, T, E, W)$ and a reachability problem $m' \in R(PN, m_0)$, a *partial solution* is a tuple $ps = (\mathcal{C}, x, \sigma, r)$, where:

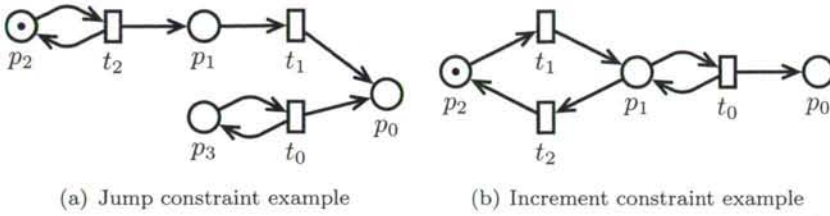


Figure 1: Example nets for jump and increment constraints

- \mathcal{C} is the set of (jump and increment) constraints, together with the state equation they define the ILP problem,
- x is the minimal solution satisfying the state equation and the constraints in \mathcal{C} ,
- $\sigma \in T^*$ is a maximal realizable firing sequence, with $\wp(\sigma) \leq x$, i.e., each transition can fire as many times as it is included in the solution vector x and if it is enabled it must fire,
- $r = x - \wp(\sigma)$ is the remainder vector.

Generating partial solutions. Partial solutions can be produced from a solution vector x (and a constraint set \mathcal{C}) by firing as many transitions as possible. For this purpose, the algorithm uses a “brute force” method. The algorithm builds a tree with markings as nodes and occurrences of transitions as edges. The root of the tree is the initial marking m_0 , and there is an edge labeled by t between nodes m_1 and m_2 if $m_1[t]m_2$ holds. On each path leading from the root of the tree to a leaf, each transition t_i can occur at most $x(t_i)$ times. Each path to a leaf represents a maximal firing sequence, thus a new partial solution. Even though the tree can be traversed only storing one path in the memory at a time using depth-first search, the size of the tree can grow exponentially. Some optimizations to reduce the size of the tree are presented later in this section.

A partial solution is called a *full solution* if $r = 0$ holds, thus $\wp(\sigma) = x$, which means that σ realizes the solution vector x . For each realizable solution x of the state equation there exists a full solution [13]. This full solution can be reached by continuously expanding the minimal solution of the state equation with constraints.

Consider now a partial solution $ps = (\mathcal{C}, x, \sigma, r)$, which is not a full solution, i.e., $r \neq 0$. This means that some transitions could not fire enough times. There are three possible situations in this case:

1. x may be realizable by another firing sequence σ' , thus a full solution $ps' = (\mathcal{C}, x, \sigma', 0)$ exists.
2. By adding jump constraints, greater, but pairwise incomparable solutions can be obtained.

3. For transitions $t \in T$ with $r(t) > 0$ increment constraints can be added to increase the token count in the input places of t , while the final marking m' must be unchanged. This can be achieved by adding new T-invariants to the solution. These T-invariants can “borrow” tokens for transitions in the remainder vector.

2.2.2 Generating constraints

Jump constraints. Each base vector of the solution space can be reached by continuously adding jump constraints to the minimal solution [13]. In order to reach non-base solutions, increment constraints are needed, but they might conflict with previous jump constraints. Jump constraints are only needed to obtain a different base solution vector. However, after the computation of the base solution, jump constraints can be transformed into equivalent increment constraints [13].

Increment constraints. Let $ps = (\mathcal{C}, x, \sigma, r)$ be a partial solution with $r > 0$. This means that some transitions (in r) could not fire enough times. The algorithm uses a heuristic to find the places and number of tokens needed to enable these transitions. If a set of places actually needs n ($n > 0$) tokens, the heuristic estimates a number from 1 to n . If the estimate is too low, this method can be applied again, converging to the actual number of required tokens. The heuristic consists of the following three steps:

1. First, the algorithm builds a dependency graph [11] to collect the transitions and places that are of interest. These are transitions that could not fire, and places that disable these transitions. Each source SCC¹ of the dependency graph has to be investigated, because it cannot get tokens from other components. Therefore, an increment constraint is needed.
2. The second step is to calculate the minimal number of missing tokens for each source SCC. There are two sets of transitions, $T_i \subseteq T$ and $X_i \subseteq T$. If one transition in T_i becomes fireable, it may enable all the other transitions of the SCC, while transitions in X_i cannot activate each other, therefore their token shortage must be fulfilled at once.
3. The third step is to construct an increment constraint c for each source SCC from the information about the places and their token requirements. These constraints will force transitions (with $r(t) = 0$) to produce tokens in the given places. Since the final marking is left unchanged, a T-invariant is added to the solution vector.

When applying the new constraint c , three situations are possible depending on the T-invariants in the Petri net:

¹Source strongly connected component, i.e., one without incoming edges from other components.

- If the state equation and the set of constraints become infeasible, this partial solution cannot be extended to a full solution, therefore it can be skipped.
- If the ILP solver can produce a solution $x + y$ (with y being a T-invariant), new partial solutions can be found. If none of them helps getting closer to a full solution, the algorithm can get into an infinite loop, but no full solution is lost. A method to avoid this non-termination phenomenon will be discussed later in this section.
- If there is a new partial solution ps' where some transitions in the remainder vector could fire, this method can be repeated.

Theorem 1. (*Reachability of solutions*) [13] *If the reachability problem has a solution, a realizable solution of the state equation can be reached by continuously adding constraints, transforming jumps before increments.*

2.2.3 Optimizations

Wimmel and Wolf [13] also presented some methods for optimization. The following are important for our work:

- **Stubborn set:** The stubborn set method [11] investigates conflicts, concurrency and dependencies between transitions, and reduces the search space by filtering the transitions. The stubborn set method usually leads to a search tree with lower degree.
- **Subtree omission:** When a transition has to fire more than once ($x(t) > 1$), the stubborn set method may not provide an efficient reduction. The same marking is often reached by firing sequences that are only different in the order of transitions. During the abstraction refinement, only the final marking of the firing sequence is important. If a marking m' is reached by firing the same transitions as in a previous path, but in a different order, the subtree after m' was already processed. Therefore, it is no longer of interest.
- **Filtering T-invariants:** After adding a T-invariant y to the partial solution $ps = (\mathcal{C}, x, \sigma, r)$, all the transitions of y may fire without enabling any transition in r , yielding a partial solution $ps' = (\mathcal{C}', x + y, \sigma', r)$. The final marking and remainder vector of ps' is the same as in ps , therefore the same T-invariant y is added to the solution vector again, which can prevent the algorithm from terminating. However, during firing the transitions of y , the algorithm could get closer to enabling a transition in r . These intermediate markings should be detected, and be used as new partial solutions.

3 Theoretical results

In this section we present our theoretical results with regard to the correctness and completeness of the original algorithm.

3.1 Correctness

Although Theorem 1 states that a realizable solution can be reached using constraints, we found that in some special cases the heuristic used for generating increment constraints can overestimate the required number of tokens for proving reachability. We prove the incorrectness by a counterexample, for which the original algorithm [13] gives an incorrect answer.

Consider the Petri net in Figure 2 with the reachability problem $(0, 1, 0, 0, 1, 0, 0, 2) \in R(PN, (1, 0, 0, 0, 0, 0, 0, 2))$, i.e., we want to move the token from p_0 to p_1 and p_4 . The example was constructed so that the target marking is reachable by the firing sequence $\sigma_s = (t_1, t_2, t_0, t_5, t_6, t_3, t_7, t_4)$, realizing the solution vector $x_s = (1, 1, 1, 1, 1, 1, 1, 1)$.

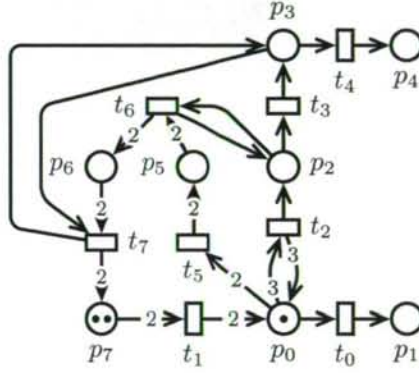


Figure 2: Counterexample for correctness

The CEGAR algorithm does the following steps. First, it finds the minimal solution vector $x_0 = (1, 0, 1, 1, 1, 0, 0, 0)$, i.e., it tries to fire the transitions t_0, t_2, t_3, t_4 . From these transitions only t_0 is enabled, therefore the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = (t_0), r_0 = (0, 0, 1, 1, 1, 0, 0, 0))$. At this point the algorithm looks for an increment constraint. The dependency graph contains transitions t_2, t_3, t_4 (since they could not fire) and places p_0, p_2, p_3 (because they disable the previous transitions). The only source SCC is the set containing one place p_0 with zero tokens (because t_0 has consumed one token from there). The algorithm estimates that three tokens are needed in p_0 , where only t_1 can produce tokens. Therefore, the T-invariant $\{t_1, t_5, t_6, t_7\}$ is added twice to the solution vector. This invariant is constructed so that for each of its firing, a token has to be produced in places p_2, p_3, p_4 , which token can no longer be removed. In the target marking only one token can be present in these places, therefore the algorithm cannot find any realizable solution, which yields the incorrect answer “not reachable”.

Notice that the problem is the over-estimation of tokens required in p_0 . Without forcing t_0 to fire, the algorithm could get a better estimation. This would imply that the invariant $\{t_1, t_5, t_6, t_7\}$ is added only once to the solution vector, producing the realizable solution x_s . The problem is that the algorithm always tries to find

maximal firing sequences, though some transitions would not be practical to fire (t_0 in the example above). Due to this, the estimated number of tokens needed in the final marking of the firing sequence may not be correct.

3.1.1 Detecting over-estimation

Our improved algorithm counts the maximal number of tokens in each place during the firing sequence of the partial solution into a vector m_{max} . If the final marking is not the maximal regarding a SCC, the algorithm might have over-estimated the required number of tokens. This can be detected by ordering the intermediate markings. Formally: an over-estimation can occur if a place p exists in a SCC, for which $m_{max}(p) > m'(p)$ holds, where m' is the final marking of the firing sequence. If such situation occurs and we do not find a full solution, we say that the problem cannot be decided. Moreover, we also developed a new method that tries to find solutions in such situations. Our first idea was to forget the original estimation (n) and estimate one instead. However, we found that over-estimation is not a problem in most cases: the algorithm still finds a realizable solution, but not the minimal. Estimating one means a slow convergence to the actual number of missing tokens, so at first we always try with the estimation n , but if no full solution is found under that subtree, we backtrack and start a new search with $n = 1$. This new approach can handle the counterexample presented in Figure 2. After no full solution is found by adding the T-invariant $\{t_1, t_5, t_6, t_7\}$ twice, we backtrack to ps_0 and try to produce only one token in p_0 . This implies that the $\{t_1, t_5, t_6, t_7\}$ is added only once to the solution vector, yielding the realizable solution x_s .

This way we can not only detect the possibility of over-estimation, but we can also find the solution in most cases. However, this method also has some limitations, which we present with the following example. Consider the Petri net in Figure 3 with the reachability problem $(1, 0, 1, 1) \in R(PN, (0, 1, 0, 1))$, i.e., moving the token from p_1 to p_2 and producing a token in p_0 . A possible solution is the vector $x_s = (1, 1, 1, 1)$, realized by the firing sequence $\sigma_s = (t_3, t_0, t_1, t_2)$.

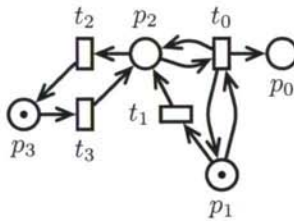


Figure 3: Example on the limitations of the new approach

The algorithm does the following steps. It finds that the minimal solution is $x_0 = (1, 1, 0, 0)$, i.e., firing t_0 and t_1 . Only t_1 is enabled, thus one partial solution $ps_0 = (\emptyset, x_0, \sigma_0 = (t_1), r_0 = (1, 0, 0, 0))$ can be found. The marking reached by σ_0 is $(0, 0, 1, 1)$, where $n = 1$ token is missing from p_1 (to enable t_0). None of the

transitions can produce tokens in p_1 , so the algorithm cannot find any constraint. The algorithm detects over-estimation because p_1 had one token before firing t_1 . Even so, a new search cannot be started, since the original estimation is also $n = 1$. The problem is that the heuristic tries to produce tokens in a place (p_1), which lacks tokens in the final marking, but had the required number of tokens at some point of the firing sequence (σ_0). Without forcing t_1 to fire, a token would be missing from p_2 , where the T-invariant $\{t_2, t_3\}$ could help. Finding the solution in such situations is an aim of our future work.

3.2 Completeness

To our best knowledge, the completeness of the algorithm has neither been proved nor disproved yet. When we examined the iteration strategy of the abstraction loop, we found a whole subclass of nets that cannot be solved with this strategy. As an example, consider the Petri net in Figure 4 with the reachability problem $(1, 1, 0, 0) \in R(PN, (0, 1, 0, 0))$, i.e., we want to produce a token in p_0 . We constructed the net so that the firing sequence $\sigma_s = (t_1, t_4, t_2, t_3, t_3, t_0, t_1, t_2, t_5)$ solves the problem. The main concept of this example is that we lend an extra token in p_1 indirectly using the T-invariant $\{t_4, t_5\}$.

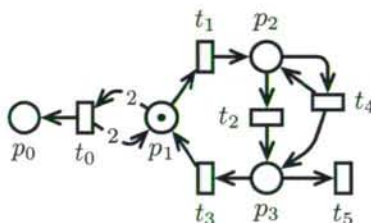


Figure 4: Counterexample of completeness

When applying the algorithm on this problem, the minimal solution vector is $x_0 = (1, 0, 0, 0, 0, 0)$, i.e., firing t_0 . Since t_0 is not enabled, the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0, 0))$. The algorithm finds that an additional token is required in p_1 and only t_3 can satisfy this need. With an increment constraint $c_1 : |t_3| \geq 1$, the T-invariant $\{t_1, t_2, t_3\}$ is added to the new solution vector $x_1 = (1, 1, 1, 1, 0, 0)$, giving us one partial solution $ps_1 = (\{c_1\}, x_1, \sigma_1 = (t_1, t_2, t_3), r_1 = r_0)$. Firing the invariant $\{t_1, t_2, t_3\}$ does not help getting closer to enabling t_0 , since no extra token can be “borrowed” from the previous T-invariant. The iteration strategy of the original algorithm does not recognize the fact that an extra token could be produced in p_3 (using t_4) and then moved in p_1 , therefore it cannot decide reachability.

4 Algorithmic contributions

In this section we present our algorithmic contributions. In Section 4.1 we show some classes of problems, for which the original algorithm cannot decide reachability, but our improved algorithm solves these problems. In Section 4.2 we present two extensions of the algorithm, solving submarking coverability problems and handling Petri nets with inhibitor arcs.

4.1 Improvements

In the previous section we proved that the algorithm is incomplete, but during our work we found some opportunities to extend the set of decidable problems. Moreover, we developed a new termination criterion, which we prove to be correct, i.e., no realizable solution is lost using this criterion.

4.1.1 New ordering of the intermediate markings

When a partial solution $ps = (\mathcal{C}, x, \sigma, r)$ is skipped using the T-invariant filtering optimization, the original algorithm checks if it was closer to enabling a transition t in the remainder during the firing sequence σ . This is done by “counting the minimal number of missing tokens for firing t in the intermediate markings occurring”[13]. We found that this criterion is not general enough: in some cases the total number of missing tokens may not be less, but they are missing from different places, where additional tokens can be produced. In our new approach, we use the following definition:

Definition 1. *An intermediate marking m_i is considered to be better than the final marking m' , if there is a transition $t \in T, r(t) > 0$ and place p with $(p, t) \in E$, for which the following criterion holds:*

$$m'(p) < w^-(p, t) \quad \wedge \quad m_i(p) > m'(p). \quad (1)$$

The left inequality in the expression means that in the final marking t is disabled by the insufficient amount of tokens in p . This condition is important, because we do not want to consider places that already have enough tokens to enable t . The right inequality means that p has more tokens in the intermediate marking m_i compared to the final marking m' .

Theorem 2. *Definition 1 is a total ordering between the intermediate markings occurring in the firing sequence σ of a partial solution and the final marking reached by σ .*

Proof. We first show that Definition 1 includes the original ordering of the intermediate markings. When the original criterion holds, the total number of missing tokens for enabling t at the marking m_i is less than at m' . This means that at least one place p must exist, which disables t , but $m_i(p) > m'(p)$, thus (1) must hold. Furthermore, Definition 1 also recognizes markings that are pairwise incomparable, because if there is at least one place p with lesser tokens missing, (1) holds. \square

Corollary 1. *The new ordering of intermediate markings extends the set of decidable problems.*

Definition 1 is more general than the original criterion, hence it does not reduce the set of decidable problems. On the other hand, we give an example when the original criterion prevents the algorithm from finding the solution. Consider the Petri net in Figure 5 with the reachability problem $(1, 0, 0, 1) \in R(PN, (0, 1, 0, 1))$, i.e., moving the token from p_1 to p_0 . The minimal solution vector is $x_0 = (1, 0, 0, 0, 0)$, i.e., firing t_0 , which is disabled by p_2 , therefore the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0))$. The algorithm looks for increment constraints and finds that only t_1 can produce tokens in p_2 . Consequently, the T-invariant $\{t_1, t_2\}$ is added to the solution vector $x_1 = (1, 1, 1, 0, 0)$. There is one partial solution $ps_1 = (\{|t_1| \geq 1\}, x_1, \sigma_1 = (t_1, t_2), r_1 = r_0)$ for x_1 , where the T-invariant is fired, but t_0 still could not fire. This partial solution is skipped by the T-invariant filtering optimization, and in all of the intermediate markings of σ_1 , totally one token is missing from the input places of t_0 . By using the original criterion, the algorithm terminates, leaving the problem as undecided. By using Definition 1, less tokens are missing from p_2 after firing t_1 than in the final marking. Continuing from here, t_0 is disabled by p_1 , where t_3 can produce tokens, therefore the T-invariant $\{t_3, t_4\}$ is added to the new solution vector $x_2 = (1, 1, 1, 1, 1)$. A full solution is found for x_2 by the realizable firing sequence $\sigma_2 = (t_1, t_3, t_0, t_2, t_4)$.

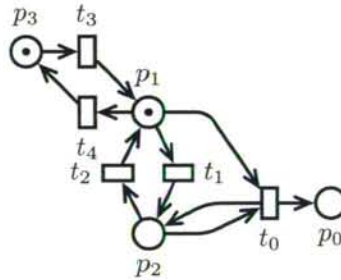


Figure 5: Example net depicting the usefulness of the new ordering

4.1.2 T-invariant filtering and subtree omission

Using T-invariant filtering and subtree omission optimizations together can prevent the algorithm from finding realizable solutions. The order of transitions in the firing sequence of a partial solution does not matter, except in one case. When a partial solution is skipped, the algorithm checks for an intermediate marking that was closer to firing a transition in the remainder vector. By using subtree omission, intermediate markings can be lost.

As an example consider the Petri net in Figure 6 with the reachability problem $(1, 0, 0, 0, 3) \in R(PN, (0, 0, 0, 0, 3))$, i.e., we want to produce a token in p_0 . A

possible solution is the vector $x_s = (1, 1, 1, 2, 2, 3, 3)$ realized by the firing sequence $\sigma_s = (t_6, t_6, t_6, t_4, t_4, t_2, t_0, t_1, t_3, t_3, t_5, t_5, t_5)$.

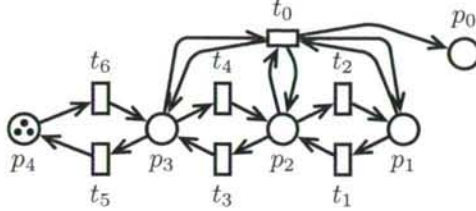


Figure 6: An example where the order of transitions matter

Here we present only the interesting points during the execution of the algorithm. As a minimal solution, the algorithm tries to fire t_0 , but it is disabled by the places p_1, p_2, p_3 . The algorithm searches for increment constraints. All the three places are in different SCCs, so the algorithm first tries to enable t_0 by borrowing a single token for all three places. By the T-invariant $\{t_1, t_2, \dots, t_6\}$ a token is carried through places p_1, p_2, p_3 , which does not enable t_0 , but there are intermediate markings in which the enabling of t_0 is closer. Continuing from any of these intermediate markings, another token is borrowed in the places p_1, p_2, p_3 , but t_0 is not enabled yet. Here comes the different order of transitions into view:

- If the two tokens are carried through places p_1, p_2, p_3 together, there are intermediate markings that are closer to firing t_0 , because previously two tokens were missing, but now only one. Continuing from these markings a third token is borrowed in places p_1, p_2, p_3 , enabling t_0 and yielding a full solution.
- If the two tokens are carried through places p_1, p_2, p_3 separately (i.e., a token is carried through the places, while the other is left in p_4 , and this procedure is repeated), there are no intermediate markings of interest, because two tokens are still missing to enable t_0 . In this case the algorithm will not find the full solution.

The order of transitions is non-deterministic, thus it is unknown which order will be omitted. Therefore, in our approach we reproduce all the possible firing sequences without subtree omission when a partial solution is skipped, and check for intermediate markings in the full tree. Although this may yield a computational overhead in some cases, we might lose full solutions otherwise.

4.1.3 New termination criterion

We have developed a new termination criterion, which can efficiently cut the search space without losing any full solutions. When generating increment constraints for a partial solution ps , as a first step the algorithm finds the set of places $P' \subseteq P$

where tokens are needed. Then it estimates the number of tokens required (n). At this point, our new criterion checks if there exists a marking m' , for which the following inequalities hold:

$$\begin{aligned} \sum_{p_i \in P'} m'(p_i) &\geq n \\ \forall p_j \in P : m'(p_j) &\geq 0. \end{aligned} \quad (2)$$

The first inequality ensures that at least n tokens are present in the places of P' , while the others guarantee that the number of tokens in each place is non-negative. These inequalities define a submarking coverability problem. Using the ILP solver, we can check if the modified form of the state equation (which we discuss in Section 4.2.1) holds for this problem. If the state equation does not hold, it is a proof that no such marking is reachable where the required number of tokens are present in the places of P' . Thus, ps can be omitted without losing full solutions.

This approach can also extend the set of decidable problems compared to the former algorithm. Consider the Petri net in Figure 7 with the reachability problem $(1, 1, 0) \in R(PN, (1, 0, 0))$, i.e., firing t_0 to produce a token in p_1 . The algorithm would add the T-invariant $\{t_1, t_2\}$ again and again to enable t_0 . Using T-invariant filtering we cannot decide whether there is no full solution or the algorithm lost it. Using our new approach we can prove that no marking exist, where two tokens are present in p_0 , therefore no full solution exists.

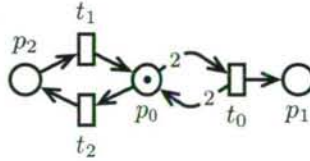


Figure 7: Example net for the new filtering criterion

4.2 Extensions

In this section we present two extensions of the CEGAR approach: solving submarking coverability problems and handling Petri nets with inhibitor arcs.

4.2.1 Submarking coverability problem

In Section 2 we introduced predicates of the form $Am' \geq b$, where A is a matrix and b is a vector of coefficients. In order to use the state equation, this condition on places must be transformed to a condition on transitions.

At first we substitute m' in the predicate $Am' \geq b$ with the state equation $m_0 + Cx = m'$, which results inequalities of the form $(AC)x \geq b - Am_0$. This set

of inequalities can be solved as an ILP problem for transitions. The extended algorithm uses this modified form of the state equation, and expands it with additional (jump or increment) constraints.

4.2.2 Petri nets with inhibitor arcs

The main problem with inhibitor arcs is that they do not appear in any form in the state equation, which is used as an abstraction. Therefore, a solution vector may be unrealizable because inhibitor arcs disable some transitions. In this case tokens must be removed from some places. Our strategy is to add transitions to the solution vector, which consume tokens from such places. Increment constraints are suitable for this purpose, but they have to be generated in a different way:

1. The first step is to construct a dependency graph similar to the original one. The graph consists of transitions that could not fire due to inhibitor arcs and places that disable these transitions. The arcs of the graph have an opposite meaning: an arc from a place to a transition means that the place disables the transition, while the other direction means that firing the transition would decrease the number of tokens in the place. Each source SCC of the graph is interesting, because tokens cannot be consumed from them by another SCC.
2. The second step is to estimate the minimal number of tokens to be removed from each source SCC. There are two sets of transitions as well, $T_i \subseteq T$ and $X_i \subseteq T$. If one transition in T_i becomes fireable, it may enable all the others in the SCC, while the needs of transitions in X_i must be fulfilled at once.
3. The third step is to construct an increment constraint for each source SCC, by firing transitions (with $r(t) = 0$) to consume the required number of tokens from the place of the SCC.

When a partial solution is not a full solution, and there are transitions disabled by inhibitor arcs, the previous algorithm is used to generate the constraint. If there are transitions disabled by normal arcs as well, both the original algorithm and the modified version must be used, taking the union of the generated constraints.

Inhibitor arcs also affect some of the optimization methods:

- Stubborn sets currently do not support inhibitor arcs.
- Using T-invariant filtering, an intermediate marking is now of interest when it has less tokens in a place, which is connected by inhibitor arc to a transition that cannot fire.
- Our new termination criterion is extended to check whether a reachable marking exists where the required number of tokens are removed.

5 Evaluation

We implemented our algorithm in the *PetriDotNet* [1] framework to evaluate its performance. The run-time results can be seen in Table 1, where TO refers to

Table 1: Measurement results for well-known benchmark problems

Model	SARA	Saturation	Our algorithm
CP_NR 10	0,2 s	-	0,5 s
CP_NR 25	111 s	-	2 s
CP_NR 50	TO	-	16s
MAPK	0,2 s	-	1 s
Kanban 1000	0,2 s	TO	1 s
FMS 1500	0,5 s	TO	5 s
SlottedRing 50	-	4 s	433 s
DPhil 50	-	0,5 s	45 s

an unacceptable run-time (> 600 seconds). The measured models are published in [4, 12, 13]. The numbers in the model names represent the parameters. We also measured a highly asynchronous, infinite state space consumer-producer model constructed by us (CP_NR in the table).

We compared our solution to the original algorithm, which is implemented in the *SARA tool* [2]. Our implementation is developed in the C# programming language, while the original is in C/C++. This causes a constant speed penalty for our algorithm. Moreover, our algorithm examines more partial solutions, which also yields computational overhead. However, the algorithmic improvements we introduced in this paper significantly reduce the computational effort for certain models (see the consumer-producer model). In addition, our algorithm can in many cases decide a problem that the original one cannot.

We also compared our algorithm to the well-known saturation-based model checking algorithm [4], implemented in our framework [12]. The lesson learned is that if the ILP solver can produce results efficiently (Kanban and FMS models), the CEGAR approach is faster by an order of magnitude than the saturation algorithm. When the size of the model makes the integer linear programming task difficult, it dominates the run-time, and saturation wins the comparison.

6 Conclusions

The theoretical results presented in this paper are twofold. On one hand, we proved the incompleteness of the iteration strategy of the original CEGAR approach by constructing a counterexample. We also presented a counterexample that proved the incorrectness of a heuristic used in the original algorithm. We corrected this deficiency by improving the algorithm to detect such situations. On the other hand, our algorithmic improvements reduce the search space, and enable the algorithm to solve the reachability problem for certain, previously unsupported classes of Petri nets. In addition, we extended the algorithm to solve two new types of problems, namely submarking coverability and handling Petri nets with inhibitor arcs. We demonstrated the efficiency of our improvements with measurements.

References

- [1] Homepage of the *PetriDotNet* framework.
<http://petridotnet.inf.mit.bme.hu/>. [Online; accessed 03-10-2014].
- [2] Homepage of the *Sara* model checker.
<http://service-technology.org/sara/index.html>. [Online; accessed 03-10-2014].
- [3] Chrzastowski-Wachtel, Piotr. Testing Undecidability of the Reachability in Petri nets with the Help of 10th Hilbert Problem. In *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*. Springer.
- [4] Ciardo, G., Marmorstein, R., and Siminiceanu, R. Saturation unbound. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 379–393. Springer, 2003.
- [5] Dantzig, George B. and Thapa, Mukund N. *Linear programming 1: introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
- [6] Esparza, Javier, Melzer, Stephan, and Sifakis, Joseph. Verification of safety properties using integer programming: Beyond the state equation, 1997.
- [7] Lipton, R.J. *The Reachability Problem Requires Exponential Space*. Research report, Yale University, Dept. of Computer Science. 1976.
- [8] Mayr, Ernst W. An algorithm for the general Petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.
- [9] Murata, Tadao. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [10] Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [11] Valmari, Antti and Hansen, Henri. Can stubborn sets be optimal? In *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 43–62. Springer, 2010.
- [12] Vörös, A., Bartha, T., Darvas, D., Szabó, T., Jámbo, A., and Horváth, Á. Parallel saturation based model checking. In *ISPDC*, Cluj Napoca, 2011. IEEE Computer Society.
- [13] Wimmel, Harro and Wolf, Karsten. Applying CEGAR to the Petri net state equation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 224–238. Springer, 2011.

Designing and Implementing Control Flow Graph for Magic 4th Generation Language

Richárd Dévai*, Judit Jász†, Csaba Nagy†, and Rudolf Ferenc†

Abstract

A good compiler which implements many optimizations during its compilation phases must be able to perform several static analysis techniques such as control flow or data flow analysis. Besides compilers, these techniques are common for static analyzers as well to retrieve information from source code, for example for code auditing, quality assurance or testing purposes. Implementing control flow analysis requires handling many special structures of the target language. In our paper we present our experiences in implementing control flow graph (CFG) construction for a special 4th generation language called Magic. While we were designing and implementing the CFG for this language, we identified differences compared to 3rd generation languages mostly because of the unique programming technique of Magic (e.g. data access, parallel task execution, events). Our work was motivated by our industrial partner who needed precise static analysis tools (e.g. for quality assurance or testing purposes) for this language. We believe that our experiences for Magic, as a representative of 4GLs, might be generalized for other languages too.

1 Introduction

Control flow analysis is a common technique to determine the control flow of a program via static analysis. The outcome of this analysis is the Control Flow Graph (CFG), which describes the control relations between certain source code elements of the application. A CFG is a directed graph: its nodes are usually basic blocks representing the statements of the code that are executed after each other without any jumps. These basic blocks are connected with directed edges representing the jumps in the control flow. A CFG is a useful tool for code optimization techniques (e.g. unreachable code elimination, loop optimization or dead code elimination). The first publications of using control flow analysis are from the 70s [1] and 80s [4, 10, 21], but since then most of the compilers have implemented this kind of analysis to construct a CFG and implement optimization phases by using it.

*FrontEndART Software Ltd, E-mail: devai@frontendart.com

†Department of Software Engineering, University of Szeged, Hungary, E-mail: {jasy,ncsaba,ferenc}@inf.u-szeged.hu

Although the basic structure of a CFG is quite common, the methods constructing it for applications are rather language dependent. Identifying control dependencies in special structures of the target language may result in special algorithms. Moreover, some program elements or applications may require minor modifications in the structure of the CFG (e.g. nodes like entry nodes).

In our paper, we present our experiences in implementing Control Flow Graph construction for a special language called Magic. This language is a so-called 4th generation language [22] because the programmer does not write source code in the traditional way, but he or she implements the application 'at a higher level' with the help of an application development environment (Magic xpa¹). This unique programming technique has many differences compared to 3GLs which are the most common languages today (Java, C, C++, C#, etc.). Due to the programming style of Magic, we had to revise traditional concepts like program components, expressions and variables during the design of a CFG for Magic applications.

The main contributions of this paper are (1) development of a CFG construction technique for applications developed in Magic xpa, (2) identification of CFG implementation differences in a 4GL context as opposed to 3GLs.

Our work was motivated by our industrial partner who needed a tool set to perform precise static analysis for code auditing and to support their testing processes. In the case of code auditing, the CFG is an important input for static code checker algorithms, while in the case of testing, the CFG is an input for algorithms which generate test scripts for automatic UI testing. Our experiences in Magic, as a representative of 4GLs could provide a good basis to implement CFG construction for other 4GLs too.

2 Related work

Control flow is a widely used information container for example in the compiler programs of 3GLs. The method of a CFG construction is well defined in [17]. We need to discover and identify the statements, and define basic blocks by selecting leader statements. Key steps are to define the structures to handle control passing, and the elements for those items of logic which implicitly influence the behavior of the control flow.

Control flow analysis has many applications, such as program transformations or source code optimizations in compilers² [11], rule checkers of analyzer tools [6, 7, 20], security checkers [5], test input generator tools³ [25], or program slicing [23]. Program dependence analysis approaches are also based upon control dependencies computed by control flow analysis [9].

The implementation of control flow analysis might differ for different languages. There are many papers published about dealing with higher-order languages (e.g. Scheme), for instance the work of Ashley et al. [2] and the PhD thesis of Ayers [3]

¹Magic Software Enterprises Website: <http://www.magicsoftware.com>

²GCC Internals Online Documentation: <http://gcc.gnu.org/onlinedocs/gccint/>

³Prasoft Products: <http://www.parasoft.com/jsp/products.jsp>

both summing up further works too [10, 21]. An extensive investigation had been done for functional languages too, which was recently summed up by Midtgaard in a survey [16].

However, CFG solutions for 4GLs are really limited. E.g. ABAP, the programming language of SAP is a popular 4GL and there are only few published flow analysis techniques which mostly deal with workflow analysis [13, 24].

In our previous work [19] we implemented a reverse engineering tool set for Magic and we found a real need to adapt some of these techniques to the language. Besides our work, Magic Optimizer⁴, as a code auditing tool also shows this necessity. This tool checks for violations of coding rules (i.e. ‘best practices’), and it is able to perform optimization checks and further analyses to give an extended overview of every part of a Magic application.

3 Specialties of a Magic Application

In the early 80’s Magic Software Enterprises introduced a 4th generation language, called Magic. The main concept was to write an application in a higher level meta language (using already existing solutions for instance for data handling and user management) and let an application generator engine create the final application. A Magic application was runnable on popular operating systems such as DOS and Unix, so applications were easily portable. Magic evolved and new versions were released, uniPaaS and lately Magic xpa. Latest releases support modern technologies such as RIA, SOA and mobile development.

The unique meta model language of Magic contains instructions at a higher level of abstraction, closer to business logic. When one develops an application in Magic, he or she actually programs the Magic Runtime Application Environment (MRE) using its meta model. This meta model is what really makes Magic a Rapid Application Development and Deployment tool.

Magic comes with many GUI screens and report editors as it was invented to develop business applications for data manipulation and reporting. The most important elements of Magic are the various entity types of business logic, namely the data tables. A table has its columns which are manipulated by a number of programs (consisting of subtasks) binded to forms, menus and help screens. These items may also implement functional logic using logic statements, e.g. for selecting variables (virtual variables or table columns), updating variables, conditional statements.

The main building blocks of a Magic application are defined in repositories. For example in the *Data Sources* repository one can define Data Objects. These are essentially the descriptions of the tables in a database. Using these objects Magic is able to handle several database management systems.

The logic of an application is implemented in the programs stored in the *Programs Repository*. Programs are the core elements of an application. These are executable entities with several sub tasks. Programs or their tasks interact with

⁴Magic xpa tools: <http://www.magic-optimizer.com/>

the user through forms to show the results of the implemented logic. Forms are also parts of tasks or programs.

Developers can edit a program with the help of the different views. The main views are the followings:

Data View. Declares which *Data Objects* are bound to the programs. The binding generally means some variable declaration, where these declarations can be real or virtual. A real declaration connects a variable to a data table column, while a virtual declaration stores some precomputed data.

Logic View. Defines *Logic Units* of a program. A task has a predefined evaluation order determined by so-called execution levels, and *Logic Units* are the parts of a task to handle the different execution levels. E.g. *Task Prefix* is the first *Logic Unit* which is executed to initialize a task. Actually, *Logic Units* are the units where the developer can write 'code' like in a 3GL. We can define statements here to perform calculations, manipulate data, call sub tasks, etc. Statements appear as *Logic Lines* in the *Logic Unit*.

Form View. Defines the properties of a window (e.g. title, size and position). Elements of a window can be typical UI elements such as controls or menus. A window is represented by a *Form Entry* in which we can use many built-in controls or our custom controls too.

As it can be seen, a Magic 4GL application differs from the programs developed in lower level languages. Developers can concentrate on implementing the business logic and the rest is done by Magic xpa.

4 Control flow graph construction

In this section, we discuss the main definitions and steps of the control flow construction for 3rd generation languages and introduce the specialties of the control flow graph construction for Magic as a representative of 4GLs.

4.1 Definitions and general steps

A **control flow graph** is a graph representation of the computation and the control flow in a program, as it can be seen in an example in Figure 1. The nodes of a CFG are basic blocks represented by rectangles. Each basic block represents a set of statements that are executed after each other sequentially. Branching can only exist at the end of blocks, after the execution of their last encapsulated statement.

The first step in the control flow creation is to determine the starting points of basic blocks [17]. These statements are called as *leaders*, and a leader can be:

- the first statement of a program,
- any statement that is the target of a conditional or unconditional branch statement,

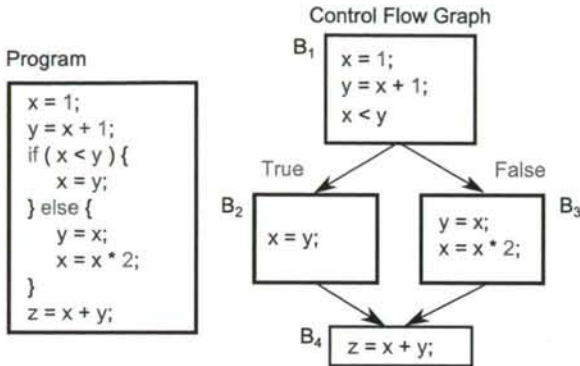


Figure 1: CFG of a simple conditional structure.

- any statement that immediately follows a conditional or unconditional branch statement,
- any statement that immediately follows a method invocation statement⁵.

If we know the sequence of statements in a program and the leaders of basic blocks, we can determine the blocks by enumerating their statements from one leader to another, but not including the next leader or the end of the program. Compilers and source code analyzers first construct an intermediate representation of the source code, called abstract syntax tree (AST) that implicitly describes the sequence of statements. With the traversal of the AST we can determine the sequence of statements, and if we want to build the control flow with finer granularity, we can examine the evaluation order of the expressions. We will discuss finer representations under the examination of Magic expressions and call types in Section 5.

In general, the control flow information of methods, procedures or the subroutines of a program are represented individually. Due to technical reasons, each of these has two special kinds of basic blocks. The *Entry* block represents the entering of a procedure, while the *Exit* block represents the returning from a called procedure. The potential control flows among procedures are represented as call edges. A connected control flow graph of a procedure with the call information gives the so-called interprocedural control flow graph (ICFG) of a program. Figure 2 shows an example of the ICFG, where call edges are represented as arrow-headed dashed lines between the call site and the *Entry* block of the called procedure, and the *Exit* block and the return statement in the caller ICFG component. In some cases, detecting procedure boundaries is not an easy task, and a call target or a branch instruction cannot be determined unambiguously. The earlier situation commonly appears in binary codes [12], while the later is typical in the presence of function pointers or virtual function calls in higher level languages. The problems appeared in 4GLs are discussed in the rest of this section.

⁵Method invocations should not be basic block boundaries in all cases only if we need compute some summarized information at the call sites in our connected application.

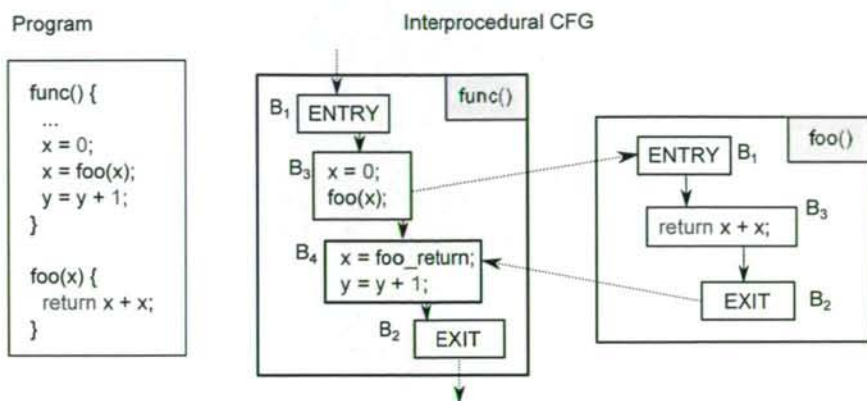


Figure 2: Example ICFG.

4.2 Challenges in Magic

Like compiler programs or other software analyzer tools do it, our first step is also to create an intermediate representation of a Magic application. We call this representation the Magic Abstract Syntax Graph (ASG) and its structure is defined by the Magic Schema [18]. The ASG allows us to traverse and process every required element of a Magic application in a well-defined hierarchical graph format through an API to determine the execution order of Magic statements. Nodes of the ASG have all the necessary attributes that can affect the control flow. E.g. ASG contains the propagation information of **Event Handlers**, which can terminate the execution of other event handlers, or the wait attribute of **Raise Event**, which determines the execution point of the given event.

Developing an application in Magic requires a unique way of thinking since the programming language is unique itself. However this programming language preserves some of the main characteristics of procedural languages. Mostly, the main logic of an application can be programmed in a procedural way via control statements in programs and their subtasks. Programs can call each other and they can call their subtasks. Also, tasks can use variables for their computations, and they can have branches within their statements. These structures of the language make it possible to adapt the CFG construction of 3GLs to Magic 4GL. For example, for every potential target of the call sites of Magic (task, event handler, developer function) we make an intraprocedural control flow graph and we connect these graphs by call edges to get the ICFG. However, there are some structures in the language which make harder to construct the CFG of an application. Here, we discuss the challenges which we face in later sections.

Tasks architecture has a special event-based execution system. There are different task types for different operations. For example, online tasks interact with the user and batch tasks run in background without any user interaction. Each task type has its own levels (e.g. task, record) and the developer can operate these with the so-called *Logic Units*. A user action or a state change in a program

can trigger predefined events that are also handled by the *Logic Units* of tasks. So, the statements (*Logic Lines*) of these *Logic Units* get executed if a certain event triggers them. The most challenging step to construct the CFG of a Magic program is to discover every circumstance that can change the flow of the control among *Logic Units* and *Logic Lines*. We have to understand and represent the effect of property changes which can influence the behavior of execution, and represent it in a well describing form.

A **Raise Event Logic Line** raises an event which is later handled by an **Event Logic Unit**. When an event is raised, the MRE immediately looks for the last available handler in the given task, and gives the control to the handler. This is the simplest case, the synchronous case. However, we could raise events asynchronously; or set the scope of handlers as they could be handled by parent tasks too, or only by the task which raised them; or every matching handler could terminate the chain of handlers if propagate property is set to 'no'. Describing the proper event handler chains within the CFG requires a complex traversal of logic units in the task hierarchy with respect to the influencing attributes. Our model is limited to those events which are raised by a code element or a form item.

Data access is supported with a rich toolset in Magic to access databases. Magic provides support to many database management systems (RDBMSs) by handling connection, transactions and generation of queries. In general, we can choose from two options to perform our transactions. In the Physical mode other DB users see our changes in RDBMS log and we use the locking system of the DB server. In the Deferred mode Magic xpa is responsible for storing our changes and committing them when we have assembled our transaction within a running task. Besides the transactional modes, we have to select the method of update process for the records we use in the transactions. Different strategies give us opportunity to handle concurrency and integrity on record updates. During the creation of the CFG we have to handle the different event handlers based on the selected transaction mode and update strategy.

Parallel task execution makes it possible to execute more programs in parallel. Parallel programs run in an isolated context where every loaded component of the main application are reloaded within the new context. In such context, a parallel program has its own copy of memory tables and its own database connections with some limitations (e.g. it cannot store data in the main program or communicate directly with other running programs). Tasks can raise asynchronous events in the context of another program to communicate, or they can use shared variables through proper functions in expressions. Parallel processes can run in Single or Multiple instance modes. In the Single mode the context is the same for each instance of the task, while the Multiple mode uses different contexts for each task. At the CFG construction we have to simulate all hidden data copying and the parallel execution of statements.

Forms have many uses during a program execution. In each case, we have to build the CFG according to the current use of forms. In a form a user can manipulate variable data, which appear in the running program as an assignment instruction, or the user can affect the running program behavior too.

5 Implementation details

The process of CFG building has several phases. First, with the traversal of the ASG we determine the sequence of statements and the evaluation order of expressions. During evaluation we collect information about calls, then we determine basic block leaders and finally, we build up the basic blocks for later processes. In our representation, each call site is a block boundary.

To determine the execution order of the contained statements and form elements of an analyzed code, we traverse its ASG from the root node step by step in the tree hierarchy and we refine the control flow information among the sub components. In each step, we define the execution order of the composed nodes of an investigated ASG node and we augment the execution sequence with additional expressions or statements, if it is needed. We do this since many semantic elements of a programming language do not appear explicitly in the source code and so in its ASG representation. Due to the hierarchical traversal, the control flow information of descendant nodes is refined after the traversal of their ancestors.

Rectangles in the figures of this section represent nodes, or groups of ASG nodes. Parallelograms denote branches where the possible flow of control depends on an attribute of **Logic Units**, **Logic Lines**, controls, variables, etc. Black arrows denote the control edges of the CFG, while dashed lines represent the call edges among the intraprocedural CFG components. Since in our representation call instructions are basic block boundaries, we represent each call with two virtual nodes called **Call Site** and **Return Site**. In some cases, we introduce solutions of alternative program versions with the help of one figure. To distinguish the variations of these versions, we use black branching points on the paths where the behaviors of the different versions are differ.

In the following sections, we discuss the cases where we could create general algorithms to process group of nodes with the same base type. Finally we introduce some special solutions where the general algorithms are not able to describe precisely the real evaluation order of the descendants of the analyzed ASG node.

5.1 General algorithms

Tasks in the ASG represent either programs or their sub tasks. The final representation of a **Task** is influenced by the implementations of its **Logic Units**, and its variables, but first we have to concentrate only on the skeleton of the tasks, since the finer control flows of **Logic Units** are determined in later steps of the traversal.

When we reach a **Task** node in the traversal, first we create an intraprocedural CFG context for the **Task** node. Our second step is to collect the sequence of logic units that take part in the execution process of the task. These nodes are the child nodes of the **Task** node in the ASG. **Task**, **Group**, and **Record** are subtypes of the **Logic Unit**, but of course, the existence of these elements are only optional in each **Task**. **Prefix** and **Suffix** are sub categories of previous **Logic Unit** subtypes controlled by an attribute. The subtype and the selected attribute value determine

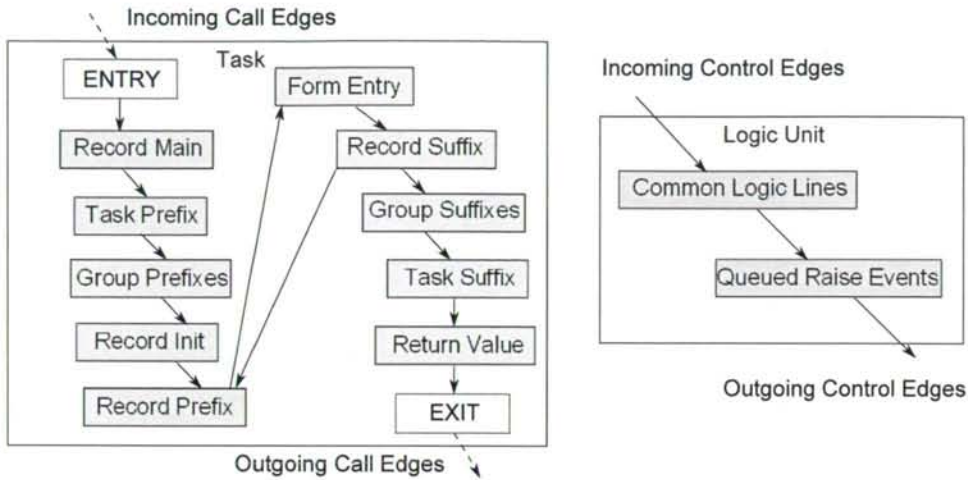


Figure 3: Evaluated control flow of a Batch Task and a Logic Unit.

the exact execution point and order of these Logic Units. So, we nominate the diversity of Logic Units with the addition of subtypes to their names as it can be seen in Figure 3.

We do not connect every Logic Unit subtype in this step, only the Task, Group and Record. For the Event and Function subtypes of the Logic Unit we associate a distinct intraprocedural CFG and handle them separately since these kinds of Logic Units can be triggered several times from distinct points.

Generated source codes and behaviors of MRE are different from the structure that we can see in Magic xpa while developing a Task, because variable declarations and initializations are also parts of the execution of logic, but defined in a separated view as we showed it in Section 3. The creations of variables and default value assignments are at the start point of a task execution. These commands are gathered by the Record Main node.

While Task and Group logic units have only two subcategories, Prefix, Suffix and Record logic units logically have three distinct in a loop of control. Each execution round of Record logic units could have an initialization part that does not appear in the code explicitly. Since it has an important effect on the control flow, we insert a virtual Record Init node into the flow of execution. If we do not find any initialization during the investigation of variables in the traversal of the record unit, or the task is not in 'write' mode and the initializations use real variables only, we can delete this Logic Unit from the CFG at the end of the traversal of the Task. During the traversal, we collect available data about Forms, their associated Controls and the logic related to them and map this information to a suitable structure of statements. This data is represented by the Form Entry node in the figure between the Record logic units as they handle the data initialization, pre- and post-processing of a record of a table. In the last step, we investigate the return expression node of the Task, and if it exits we connect it as the last item

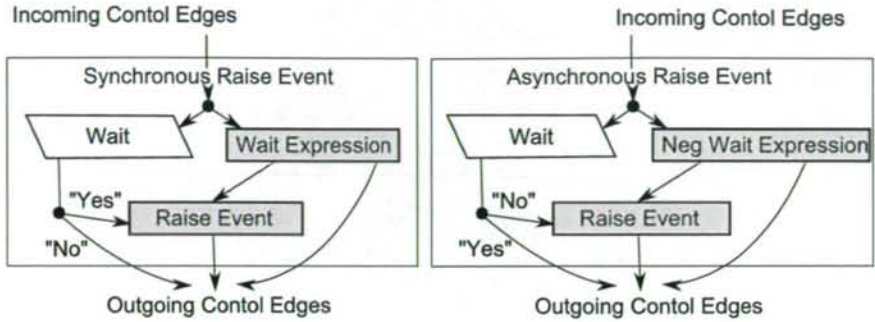


Figure 4: Control flow of Raise Events.

before the **Exit** block of the **Task**.

On the left side of Figure 3, we can see the execution order of a **Batch Task** or a **Browse Task**. This task contains variables, implements all possible **Logic Unit** subtypes, and defines a return expression.

After visiting all the nodes of a **Task**, we are able to build up its basic blocks and determine the control and call edges among them. With this information we can derive the exact execution order of the statements and expressions.

Each **Logic Unit** consists of **Logic Lines**. Generally, **Logic Lines** have two distinct kinds. First, the execution of the logic line does not depend on its properties or on the execution of other logic lines; we handle them as they can run sequentially in the order of appearance until further checks. We refer to these as **Common Logic Lines**. The second kind is the so-called **Raise Event** with an attribute called **wait** that we have to observe. With the **Raise Event** nodes we determine the asynchronously executed **Queued Raise Events** according to Figure 3, if the value of the wait attribute is 'no'. The wait attribute of the **Raise Event** can have a 'yes' or 'no' boolean constant value or the result of a boolean expression. Since the execution of these lines depend on the value of the wait attribute, we have two distinct cases. If this value is logically true the raise events are synchronous otherwise they are asynchronous. An illustration can be seen in Figure 4.

The execution of a **Logic Line** depends on a condition. If this condition evaluates to true, the flow of control goes into the statement, which describes the exact behavior of the logic line. Although this part of the evaluation of the logic lines is general, the behavior of the distinct subtypes of **Logic Lines** can be very different as we can see in the next section.

5.2 Specific algorithms

As it was mentioned in the last subsection, the **Function** and **Event Logic Unit** nodes are different from other logic units, but similar to each other. Since the execution of these units depend on their context, and their execution can be triggered from different points of the program, it is better to handle them in a similar way as we handled the **Task** nodes. Hence, for these nodes we created intraprocedural

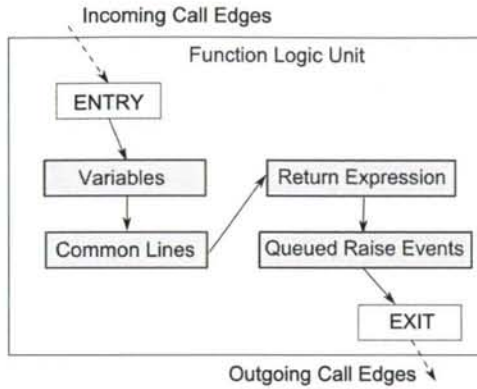


Figure 5: CFG of Function Logic Unit.

CFG representations which are callable from distinct program points. Next, we collect **Logic Lines** which are variable declarations from their contained **Logic Lines**, because they are not necessarily in order before all other **Logic Lines**, but executed collectively at the beginning of the execution of the **Logic Unit**. Next we have to perform an algorithm like we performed for **Logic Units**. The difference between **Function** and **Event Logic Units** is that the former could define a **Return Expression** declared by an attribute of the **Logic Unit** which is executed before the **Queued Raise Events** as it is shown in Figure 5.

Logic Lines are evaluated through the traversal by specific evaluators. These elements of logic are much more unique from the point of view of control flow processing than the **Tasks** and **Logic Units**. We introduce some of these to show the variety and the complexity of their processing.

A **Block** node is implemented by a **Logic Line** pair. A **While Block** with its related **End Block** declare the start and the end of the **Block**. These two encapsulate the body of the **Block**. When we find a **While Block** in the ASG, we have to search its terminating **End Block** node, because they are not connected directly in the ASG. The condition of a **While Block** can be a 'yes' or 'no' constant or an **Expression**. Nesting **Block** nodes make it harder to carry out this task. The left hand side of Figure 6 shows the evaluation of a while structure. The structure of an **If Block** is similar to the structure of a **While Block**. First, we have to search the corresponding **End Block** and **Else Blocks** for each **If Block** node. The multiple selection is implemented by the optional condition argument of an **Else Block** node.

The right side of Figure 6 shows a **Call** logic line which implements a call based on a Magic generated identifier of a program, a sub task or a public name, etc. A **Call** logic line node has an optional argument list and could receive a return value. The passed-by-reference arguments are updated after the control is given back to the **Return Site**. To implement this behavior in the CFG, we have to create update nodes for them. Before the actual call, we insert a **Call Site** node into the CFG, while after the execution of the **Exit Block** of the called CFG we

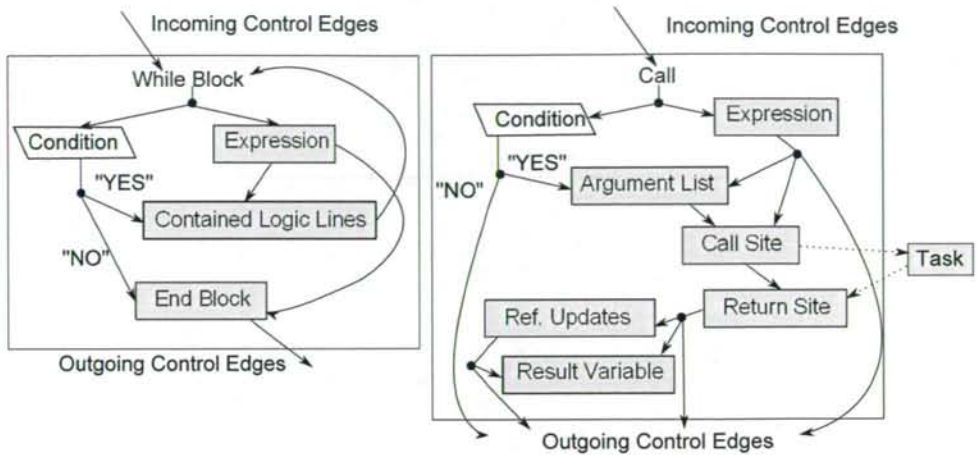


Figure 6: CFG of a While block and a general Call logic line.

nominate the return with a **Return Site** node.

Select Logic Lines defined in the Data View are separated from the code.

Semantically these **Select Logic Lines** are executed in the **Record Main** and **Record Init Logic Units** during of a given task. Hence, handling expressions of **Select Logic Lines** is similar to the way we handle normal **Logic Line** types.

All **Expressions** of Magic are arranged into subtypes by categories in our ASG representation. An **Expression** can be a literal, a unary or binary operation or a **Function Call** that refers to a built-in function or **Function Logic Units**. **Literals** can make a reference to an identifier, a resource or a component, or they can contain a constant value.

The control flow of a **Function Call** can be built-up as a simpler **Call Logic Line**, the only difference is that its arguments cannot be passed by reference.

5.3 Associated control structures of Form logic

One primary motivation of our work was to support the UI testing of Magic applications, so it is essential to represent control dependencies arising via UI elements such as controls of Forms as they provide the main interface for user interactions. Magic programs basically follow a strongly event driven model and most of the events are generated by the UI elements of Forms.

In Magic, the structure of the UI or the layout of Forms and controls is readily available in the ASG, so thanks to the language, the connection between **Controls** and their related logic is also available (e.g. relation between an edit box and its related variable; or relation between a menu and the task to be executed). Based on this information, we can extend the CFG with UI elements and their control relations, so we can get a better view of the control flow than in an event-based context.

There have been several attempts to develop techniques for the generation of

automated GUI tests with less or more success by abstract state machines, but most of the techniques are *ad hoc*, and mostly manual; in addition, there is also a great potential in modeling event interactions with directed graphs e.g. by modeling the event flows of applications as noted in [15]. Similarly, we represent events in the CFG as there is a great number of events built into Magic.

There are several control types which we group into the following two logical classes based on the control structures that we handle them with:

Group₁: **Push Buttons**, **Sub Forms**, **Menus**, Sub Menus and Context menus

Group₂: Input controls such as **Edit boxes** and Lists, Radio Buttons and Check Boxes

Group₁ contains items which are responsible for process control and embedding, while items of *Group₂* handle input data and could be wrapped into a validation context.

Sub Forms are useful to integrate a task form into the form of another task while maintaining the subform's task data handling and record cycle activities as independent of the parent task. This is a good solution for reusing data, logic and also their GUI parts.

Input controls (e.g. edit boxes) are useful for setting the value of a **Real** or **Virtual Variable**. These controls take input data to change the values of variables and there are several kinds of validators and programming logic (e.g. logic units) to handle their usage.

Menus can be used to navigate between different programs in an application through calls and events, so they provide access to a large variety of functionalities.

Push Buttons are good for triggering events to place several crucial behavior just in front of the user to ease navigation, and give an opportunity to stress the operations which should be emphasized. E.g. it is possible, but rather unusual to use a context menu in a calculator application to add numbers.

In a form the simple behaviour would be that all the controls are related to each other, since the sequence of control invocations is undetermined and we need to represent all possible sequences (e.g. imagine the user pushing the buttons randomly). Representing this would radically increase the number of edges in the CFG, so for simplicity, we introduce the so-called *entry* nodes which virtually join all the controls in a Form. A Form can have multiple exit points depending on which **Suffix** follows the Form in the program.

In Figure 7. we see the CFG part of a form structure with an example *entry* node, which we introduced before. The controls like **Edit Box** could be surrounded with different types of validation logic. In the case of *Group₂* controls, there is one-to-one correspondence established by the language for variables, so to each **Control** a **Variable** with a **Variable Logic Unit** that is responsible for handling the changes of its value will be assigned. These Logic Units either receive the original value and the new one, or only one of them and they do an initialization, validation or checking step.

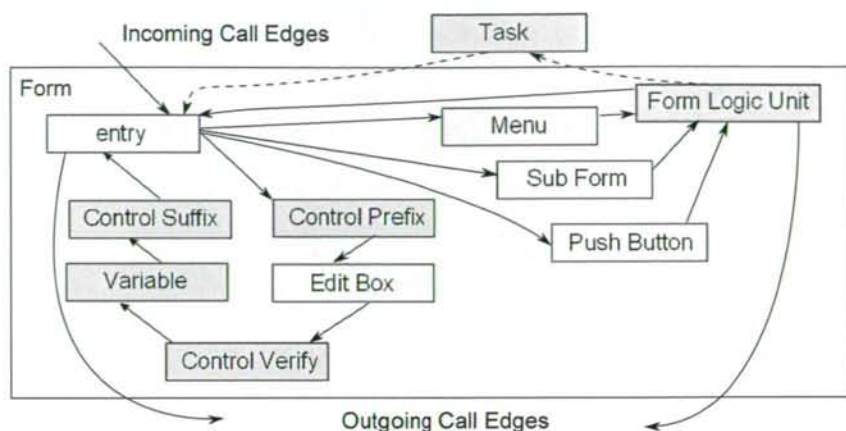


Figure 7: CFG context of a Form Entry and its encapsulated controls.

Menus, Push Buttons and Sub Forms are different as they are responsible for navigation and encapsulation. Menus can mainly possess two different types of behavior as they can trigger events, or they can call other Programs. To represent this behaviour, we created a virtual Logic Unit called **Form Logic Unit**. The purpose of this virtual logic unit is to group together virtual raise event statements and call the statements which are stored in the ASG under the corresponding task. These virtual raise event statement simulate how the control of a form can actually raise an event in the control flow (see the **Form Logic Unit** node in Figure 7). Such a logic unit can have incoming control edges from *Group₂* controls.

Push Buttons are similarly handled as **Menus** except that they cannot call other tasks only raise an event.

A **Sub Forms** is an embedded form in another main Form. The content of the Sub Form is provided by a sub task of the task of its main form. In order to represent this structure in the CFG, Sub Forms are connected through a task **Call** to their main Form.

Program control can leave the form context through events related to task termination or user actions. This is symbolized by outgoing control edges in Figure 7.

6 Evaluation

We implemented our technique in C++ and verified it through result validations and performance tests. For this verification we created a testbed with 105 Magic applications which were specifically designed to implement special control structures in Magic. We created a simple batch script to construct the ASG and then the CFG for each application in this testbed. Then, we manually compared the constructed CFGs to the program code. To perform this comparison, we exported the constructed CFG to graphML format which can be easily visualized with yED⁶.

⁶yED Graph Editor: http://www.yworks.com/en/products_yed_about.html

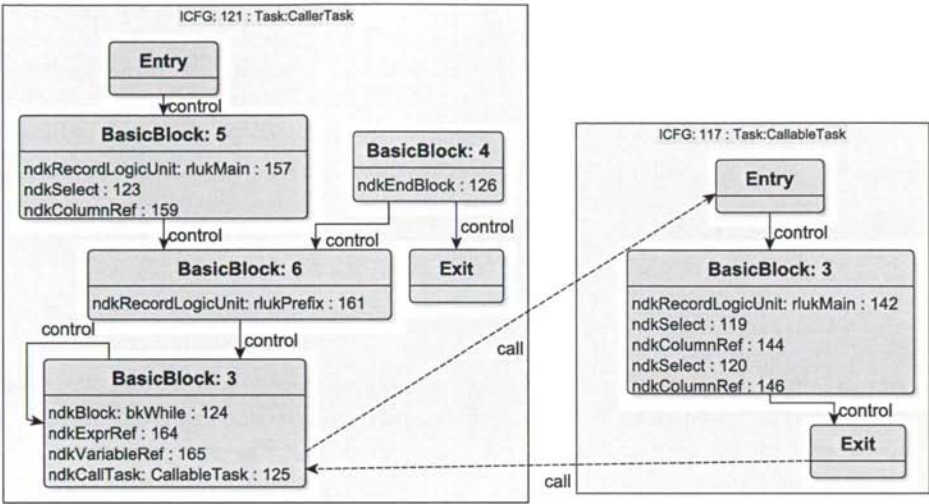


Figure 8: Visualized ICFG by generated graphML dump.

An exported picture of a sample graphML can be seen in Figure 8. The original code contains an infinite `While` Block. This information is shown in the figure too, where basic block with id 4 is unreachable. This information could be easily retrieved by API calls during the traversal of the CFG. Of course, in this case this possibly malformed control structure is recognized by Magic xpa too, it warns the programmer about the existence of the infinite loop. The example of the figure contains a call from the body of the `While` Block. This call also appeared in our ICFG. We compared all the resulting dumps with the original source code manually, and we found that each ICFG gave a good description of the possible execution paths of the original code.

After manually evaluating all the constructed CFGs of the testbed, we evaluated our implementation on a larger application too (the demo application released with Magic xpa). Not only did we construct the CFG and check its consistency, but with profiling we gathered run-time statistics of our algorithms too.

To verify the usability of our algorithms, we ran our implementation on an Intel XENON E5450 @ 3GHz 32 GB Windows Server 2008. As performance results on a medium sized sample project with nearly 200.000 nodes and about 500.000 attributes we got a 0,598 seconds runtime of the ICFG computation. The ICFG computation was carried out in an affordable time, so it was adaptable in any approaches based on this information.

7 Limitations of the approach

Besides the shown advantages of our technique, there are a few limitations too. Here we describe two main limitations.

Our event handling does not handle all the possible specialties of a Magic ap-

plication. Currently, the implementation is able to follow the events that are raised and handled inside the code with a raise event statement or a certain logic unit. The internal events of Magic xpa (such as hotkeys) are not yet supported unless raised by a raise event statement.

Our recent CFG model does not support the representation of parallel task executions given by section 4.2. To improve our model, we should investigate previous work about the limitations and possible application forms of CFG for parallelism support e.g. [14].

8 Summary and Future Work

In our paper, we presented an application of CFG concepts for a specific 4th generation language, Magic 4GL. We used a static analysis approach to gain information from the generated Magic source code and to build a CFG with fine granularity. We created a reusable library for further use of our model which makes it possible to perform further analyses and process the CFG and ICFG structures which we created. We created a textual and an XML based graphML dump to make it easy to get an overview of the processed information.

Our evaluation showed that the approach implemented is applicable for middle-sized Magic applications. The method presented had an affordable space requirement and it constructed the CFG fast enough to analyze large projects too.

Besides, we showed that implementing control flow analysis for a higher-level language, such as Magic, was possible via adapting 3GL techniques, but the unique structures of the language may result in special methods and structures in the CFG as well. For example, the use of Events enabled us to gather more precise information compared to 3GLs where these structures are mostly dynamic.

Conceptually, the presented technique could be applied to other 4GLs too. The core elements of the CFG should be the same in a language independent way (e.g. UI handling), but special constructs of the language should require special solutions (e.g. events and raise event handling and the task record loop).

We applied our work in a research project which was carried out in cooperation with our industrial partner to automatically generate test cases and test input for a GUI test automation tool for Magic [8]. Additionally, we targeted the development of a trace analyser tool to support coverage measurement purposes based on our CFG solution. In this project, we created a path analyser and generator tool for traversing the CFG and generating potential execution paths for the test scripts. As we expected, the growth of path space was exponential [15], so we had to apply several pre-filtering techniques over the CFG before or during the generation, although post-filtering was also possible, it was inefficient or very limited. We created an XML based filtering technique where we could select a sub-component of the CFG with the help of the work flow descriptions of the application analyzed (a work flow described a certain functionality with its related programs and tasks). After the filtering we could execute the script generator tool to create test cases for the Magic XPA application that we analyzed. Our results were promising, hence

our CFG technique seemed to be useful in supporting automatic UI testing with test script generation and validation via test coverage measurements.

Acknowledgements

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0039.

References

- [1] Allen, Frances E. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.
- [2] Ashley, J. M. and Dybvig, R. K. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, July 1998.
- [3] Ayers, Andrew Edward. *Abstract analysis and optimization of Scheme*. PhD thesis, Cambridge, MA, USA, 1993.
- [4] Cousot, P. Semantic foundations of program analysis. In Muchnick, S.S. and Jones, N.D., editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
- [5] D., Anupam, J., Somesh, L., Ninghui, Melski, D., and Reps, T. Analysis techniques for information security. *Synthesis Lectures on Information Security, Privacy, and Trust*, 2(1):1–164, 2010.
- [6] Ferenc, Rudolf, Beszédes, Árpád, and Gyimóthy, Tibor. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.
- [7] Ferenc, Rudolf, Beszédes, Árpád, Tarkainen, Mikko, and Gyimóthy, Tibor. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.
- [8] Fritsi, Dániel, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. A layout independent GUI test automation tool for applications developed in Magic/uniPaaS. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST 2011)*, pages 248–259, 2011.
- [9] Horwitz, S., Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24(7):28–40, June 1989.

- [10] Jones, Neil D. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, UK, 1981. Springer-Verlag.
- [11] Kennedy, K. and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [12] Kiss, Ákos, Jász, Judit, Lehotai, Gábor, and Gyimóthy, Tibor. Interprocedural static slicing of binary executables. In *Proc. Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 118–127, September 2003.
- [13] Kowalkiewicz, M., Lu, R., Bäuerle, S., Krümpelmann, M., and Lippe, S. Weak dependencies in business process models. In Abramowicz, Witold and Fensel, Dieter, editors, *Business Information Systems*, volume 7 of *Lecture Notes in Business Information Processing*, pages 177–188. Springer Berlin Heidelberg, 2008.
- [14] Lam, M. S. and Wilson, R. P. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.
- [15] Memon, Atif M. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, September 2007.
- [16] Midtgaard, Jan. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.
- [17] Muchnick, Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [18] Nagy, Csaba, Vidács, László, Ferenc, Rudolf, Gyimóthy, Tibor, Kocsis, Ferenc, and Kovács, István. Complexity measures in 4GL environment. In *Proceedings of the 2011 international conference on Computational science and Its applications - Volume Part V*, pages 293–309. Springer-Verlag, 2011.
- [19] Nagy, Csaba, Vidács, László, Ferenc, Rudolf, Gyimóthy, Tibor, Kocsis, Ferenc, and Kovács, István. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 343–346, 2011.
- [20] Rech, J. and Schäfer, W. Visual support of software engineers during development and maintenance. *SIGSOFT Softw. Eng. Notes*, 32(2):1–3, March 2007.
- [21] Shivers, O. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.

- [22] The Institute of Electrical and Eletronics Engineers. IEEE standard glossary of software engineering terminology. IEEE Standard, September 1990.
- [23] Tip, Frank. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
- [24] Vanhatalo, Jussi, Völzer, Hagen, and Leymann, Frank. Faster and more focused control-flow analysis for business process models through sese decomposition. In Krämer, BerndJ., Lin, Kwei-Jay, and Narasimhan, Priya, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2007.
- [25] Visser, W. and Păsăreanu, C. S. and Khurshid, S. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

Code Coverage Measurement Framework for Android Devices*

Ferenc Horváth[†], Szabolcs Bognár[†], Tamás Gergely[†], Róbert Rácz[†],
Árpád Beszédes[‡], and Vladimir Marinkovic[‡]

Abstract

Software testing is a very important activity in the software development life cycle. Numerous general black- and white-box techniques exist to achieve different goals and there are a lot of practices for different kinds of software. The testing of embedded systems, however, raises some very special constraints and requirements in software testing. Special solutions exist in this field, but there is no general testing methodology for embedded systems. One of the goals of the CIRENE project was to fill this gap and define a general testing methodology for embedded systems that could be specialized to different environments. The project included a pilot implementation of this methodology in a specific environment: an Android-based Digital TV receiver (Set-Top-Box).

In this pilot, we implemented method level code coverage measurement of Android applications. This was done by instrumenting the applications and creating a framework for the Android device that collected basic information from the instrumented applications and communicated it through the network towards a server where the data was finally processed. The resulting code coverage information was used for many purposes according to the methodology: test case selection and prioritization, traceability computation, dead code detection, etc.

The resulting methodology and toolset were reused in another project where we investigated whether the coverage information can be used to determine locations to be instrumented in order to collect relevant information about software usability.

In this paper, we introduce the pilot implementation and, as a proof-of-concept, present how the coverage results were used for different purposes.

Keywords: coverage, embedded, traceability, Android

*Part of this work was done in the Cross-border ICT Research Network (CIRENE) project (project number is HUSRB1002/214/044) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union. This research was partially supported by the Hungarian national grant **GOP-1.1.1-11-2011-0006**.

[†]University of Szeged, Department of Software Engineering, E-mail: {ferenc,bszabi,gertom,rrobi,beszedes}@inf.u-szeged.hu

[‡]University of Novi Sad, Faculty of Technical Sciences, E-mail: vladam@uns.ac.rs

1 Introduction

Software testing is a very important quality assurance activity of the software development life cycle. With testing, the risk of a residing bug in the software can be reduced, and by reacting to the revealed defects, the quality of the software can be improved. Testing can be performed in various ways. For example, static testing includes the manual checking of documents and the automatic analysis of the source code without executing the software. During dynamic testing the software or a specific part of the software is executed. Many dynamic test design techniques exist, the two most well known groups among them are black-box and white-box techniques.

Black-box test design techniques concentrate on testing functionalities and requirements by systematically checking whether the software works as intended and produces the expected output for a specific input. The techniques take the software as a black box, examine “what” the program does without having any knowledge on the structure of the program, and they are not intrerested in the question “how?”.

On the other hand, white-box testing examines the question “How does the program do that?”, and tries to exhaustively examine the code from several aspects. This exhaustive examination is given by a so-called coverage criterion which defines the conditions to be fulfilled by the set of statement sequences executed during the tests. For example, 100% instruction coverage criterion is fulfilled if all instructions of the program are executed during the tests. Coverage measures give a feedback on the quality of the tests themselves.

The reliability of the test can be improved by combining black-box and white-box techniques. During the execution of test cases generated from the specifications using black-box techniques, white-box techniques can be used to measure how completely the actual implementation is checked. If necessary, reliability of the tests can be improved by generating new test cases for the code fragments not verified.

1.1 Specific problems with embedded system testing

Testing in embedded environments has special attributes and characteristics. Embedded systems are neither uniform nor general-purpose. Each embedded system has its own hardware and software configuration typically designed and optimized for a specific task, which affects the development activities on the specific system. Development, debugging, and testing are more difficult since different tools are required for different platforms. However, high product quality and testing that ensures this high quality is very important. Suppose that the software of a digital TV with play-from-USB capabilities fails to recover after opening a specific media file format and this bug can only be repaired by replacing the ROM of the TV. Once the TV sets are produced and sold, it might be impossible to correct this bug without spending a huge amount of money on logistic issues. Although there are some solutions aiming at the uniformisation of the software layers of embedded systems (e.g., the Android platform [12]), there has not been a uniform methodology for embedded systems testing.

1.2 The CIRENE project

One of the goals of the CIRENE project [19] was to define a general testing methodology for embedded systems that copes with the above mentioned specialities and whose parts can be implemented on specific systems. The methodology combines black-box tests responsible for the quality assessment of the system under test and white-box tests responsible for the quality assessment of the tests themselves. Using this methodology the reliability of the test results and the quality of the embedded system can be improved. As a proof-of-concept, the CIRENE project included a pilot implementation of the methodology for a specific, Android-based digital Set-Top-Box system. Although the proposed solution was developed for a specific embedded environment, it can be used for all Android-based embedded devices such as smart phones or tablets.

The coverage measurement toolchain plays an important role in the methodology (see Figure 1). Many coverage measurement tools (e.g., EMMA [28]) exist that are not specific but can be used on Android applications. However, these are applicable only during the early development phases as they are able to measure code coverage on the development platform side. This kind of testing omits to test the real environment and misses the hardware-software co-existence issues which can be essential in embedded systems. We are not aware of any common toolchain that measures code coverage directly on Android devices.

Our coverage measurement toolchain starts with the instrumentation of the application under test, which allows us to measure code coverage of the given application during test execution. As the device of the pilot project runs the Java-based Android operation system, Java instrumentation techniques can be used. Then, the test cases are executed and the coverage information is collected. In the pilot implementation, the collection is split between the Android device and the used testing tool RT-Executor [24]: the service collects the information from the individual applications of the device, while the testing tool processes the information (through its plug-ins).

The coverage information gathered with the help of the coverage framework can be utilized by many applications in the testing methodology. They can be used for selecting and prioritizing test cases for further test executions, or for helping to generate additional test cases if the coverage is not sufficient. It is also useful for dead code detection or traceability links computation.

The resulting methodology and toolset were reused in another project which aims usability testing on Android devices. In this project, we investigated whether the coverage information gathered by the described method can be used to determine locations in the code that must be “watched” during test executions in order to collect relevant information of the usability of the software. The long-term goal was to reduce the number of instrumentation points in the examined software which results in less performance decrease and, thus, supports aimed mass field testing.

In this paper, we introduce the pilot implementation, discuss our experiments conducted to examine the further use of the coverage results, and evaluate these experiments.

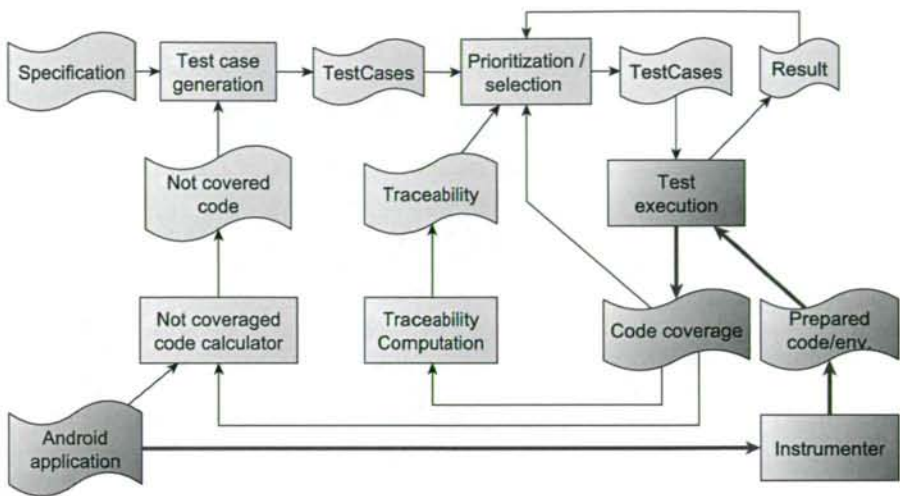


Figure 1: Coverage collection methodology on the Set-Top-Box

1.3 Paper structure

The rest of the paper is organized as follows. In Section 2 we give an overview on the related work. Section 3 presents the implementation of the coverage measurement framework. In Section 4 some use cases are shown to demonstrate the usefulness of coverage information. Finally, we summarize our achievements and introduce some possible future works in the last section.

2 Related Work

In the CIRENE project, one of our first tasks was to assess the state-of-the-art in embedded systems testing techniques with special attention to the combined use of black and white-box techniques. As a result of this task we presented a technical report [3] in which we report only a few number of combined testing techniques that have been specialized and implemented in the embedded environment.

Gotlieb and Petit [17] presented a path-based test case generation method. They used symbolic program execution and did not execute the software on the embedded device prior to the test case definitions. We use code coverage measurement of real executions to determine information that can be used in test case generation.

José et al. [9] defined a new coverage metric for embedded systems to indicate instructions that had no effect on the output of the program. Their implementation used source code instrumentation and worked for C programs at instruction level, and had a great influence on the performance of the program. Biswas et al. [4] also utilized C code instrumentation in embedded environment to gather profiling

information for model-based test case prioritization. We use binary code instrumentation at method level, use traditional metric that indicates whether the method is executed during the test case or not, and our solution has a minimal overhead on execution time. The resulting coverage information can also be used for test case selection and prioritization.

Hazelwood and Klauser [18] worked on binary code instrumentation for ARM-based embedded systems. They reported the design, implementation and applications of the ARM port of the Pin, a dynamic binary rewriting framework. However, we are working with Android systems that hides the concrete hardware architecture but provides a Java-based one.

There are many solutions for Java code coverage measurement. For example, EMMA [28] provides a complete solution for tracing and reporting code coverage of Java applications. However, it is not concerning the specialities of Android or any embedded systems.

Most of the coverage measurement tools utilize code instrumentation. In Java-based systems, byte code instrumentation is more popular than source code instrumentation. There are many frameworks providing instrumenting functionalities (e.g., DiSL [21], InsECT [6, 26], jCello [27], and BCEL [2]) for Java. These are very similar to each other regarding their provided functionalities. We chose *Javasist* [7] to be our instrumentation framework in the pilot project.

Traceability links between requirements and source code are important in software development. Automatic methods for traceability link detection include information retrieval ([20, 1, 8]) and probabilistic feature location ([22]) and combined techniques ([11]). We used code coverage based feature location to retrieve traceability information.

3 Coverage Measurement Toolchain

The implemented coverage measurement toolchain consists of several parts. First, the applications selected for measurement have to be prepared. This process includes program instrumentation that inserts extra code into the application so that the application can produce the information necessary for tracing its execution path during the test executions. The modified applications and the environment that helps collecting the results must be installed on the device under test.

Next, tests are executed using this measurement environment and the prepared applications, and coverage information is produced. In general, test execution can be either manual or automated. In the current implementations, we use two different approaches for test automation.

Within the CIRENE pilot implementation *RT-Executor* [24] (a black-box test automation tool for multimedia devices testing) is used as the automation tool.

In the usability testing project we use a simplified tool in the testing process, which helps gathering and preparing the coverage information for the evaluation. The functions of this tool are based on the *Robotium* [16] framework. Robotium is an Android test automation framework that has full support for native and hybrid

applications and makes it easy to write powerful and robust automatic black-box tests for Android applications.

During the execution of the test cases, the instrumented applications produce their traces which are collected, and coverage information is sent back to the automation tool.

Third, the coverage information resulted from the previous test executions is processed and used for different purposes, e.g., for test selection and prioritization, additional test case generation, traceability computation, and dead code detection.

In the rest of this section, we describe the technical details of the coverage measurement toolchain.

3.1 Preparation

In order to measure code coverage, we have to prepare the environment and/or the programs under test to produce the necessary information on the executed items of the program. In our case, the Android system uses the Dalvik virtual machine to execute the applications. Although modifying this virtual machine to produce the necessary information would result in a more extensive solution that would not require the individual preparation of the measured applications, we decided not to do so, as we assumed that modifying the VM itself had higher risks than modifying the individual applications. With individual preparation it is much easier to decide what to measure and at what level of details. So, we decided to individually prepare the applications to be measured. As we were interested in method level granularity, the methods of the applications were instrumented before test execution, and this instrumented version of the application was installed on the device. In addition, a service application serving as a communication interface between the tested applications and the network was also necessary to be present on the device.

3.1.1 Instrumentation

During the instrumentation process, extra instructions are inserted in the code of the application. These extra instructions provide additional functionality (e.g., logging necessary information) but they should not modify the original behaviour of the application. Instrumentation can be done on the source code or on the binary code.

In our pilot implementation, we are interested in method level code coverage measurement. It requires the instrumentation of each method inserting a code that logs the fact that the method is called. As our targets are Android applications usually available in binary form, we have chosen binary instrumentation.

Android is a Java-based system which in our case means that the applications are written in Java language and compiled to Java Bytecode before a further step creates the final Dalvik binary form of the Android application. The transformation from Java to Dalvik is reversible, so we can use Java tools to manipulate the

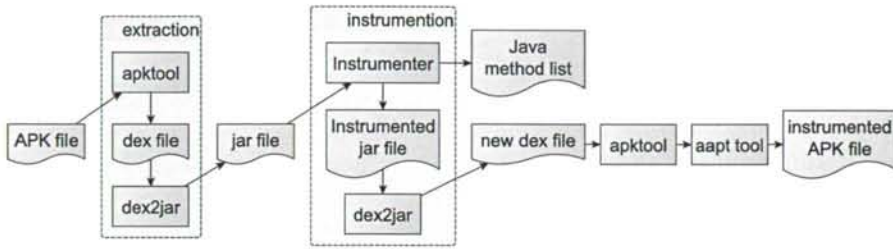


Figure 2: Instrumentation toolchain

program and instrument the necessary instructions. We used the *Javassist* [7] library for Java bytecode instrumentation, *apktool* [13] for unpacking and repacking the Android applications, the *dex2jar* [14] tool for converting between the Dalvik and the Java program representations, and *aapt* [15] tool to sign the application. The *Instrumentation toolchain* (see Figure 2) is the following:

- The Android binary form of the program needs to be instrumented. It is an *.apk* file (a special Java package, similar to the *.jar* files, but extended with other data to become executable).
- Using the *apktool* the *.apk* file is unpacked and *.dex* file is extracted. This *.dex* file is the main source package of the application, it contains its code in a special binary format. [15, 5]
- For all *.dex* files the *dex2jar* is used to convert them to *.jar* format.
- On the *.jar* files we can use the *JInstrumenter*. The *JInstrumenter* is our Java instrumentation tool based on the *Javassist* library [7].

JInstrumenter first adds a new collector class with two responsibilities to the application. On the one hand, it contains a coverage array that holds the numbers indicating how many times the methods (or any other items that is to be measured) were executed. On the other hand, this class is responsible for the communication with the service layer of the measurement framework. Next, the *JInstrumenter* assigns a unique number as ID to each of the methods. This number indicates the method's place in the coverage array of the collector class. Then a single instruction is inserted in the beginning of all methods which updates the corresponding element of the coverage array on all executions of the method.

The result of the instrumentation is a new *.jar* file with instrumented methods and another file with all the methods' names and IDs.

- The instrumented *.jar* files are converted to *.dex* files using the *dex2jar* tool again.

- Finally, the `.apk` file instrumented application is created by repacking the `.dex` files with the `apktool` and signing it with the `aapt` tool.

During the instrumentation, we give a name to each application. This name will uniquely identify the application in the measurement toolchain, so the service application can identify and separate the coverage information of different applications.

After the instrumentation, the application is ready for installation on the target device.

3.1.2 Service application

In our coverage measurement framework implementation it is necessary to have an application that is continuously running on the Android device in parallel with the program under test. During the test execution, this application is serving as a communication interface between the tested applications and the external tool collecting and processing the coverage data. On the one hand this is necessary because of the rights management of the Android systems. Using the network requires special rights from the application and it is much simpler and more controllable to give these rights to only a single application than to all of the tested applications. On the other hand, this solution provides a single interface to query the coverage data even if there are more applications tested and measured simultaneously.

In Android systems, there are two types of applications: “normal” and “service”. Normal applications are active only when they are visible. They are destroyed when moved in the background, although their state can be preserved and restored on the next activation. Services are running in the background continuously and are not destroyed on closing. So, we had to implement this interface application as a service. It serves as a bridge between the Android applications under test and the “external world” as it can be seen on Figure 3. The tested applications are measuring their own coverage and the service queries these data on-demand. As the communication is usually initiated before the start and after the end of the test cases, this means no regular communication overhead in the system during the test case executions.

Messages are accepted from and sent to the external coverage measurement tools. The communication uses JSON [10] objects (type-value pairs) over the TCP/IP protocol. Implemented messages are:

NEWTC The testing tool sends this message to the service to sign that there is a new test case to be executed and asks it to perform the required actions.

ASK The testing tool sends this message to query the actual coverage information.

COVERAGE DATA The service sends this message to the testing tool in response to the **ASK** message. The message contains coverage information.

Internally, the service also uses JSON objects to communicate with the instrumented applications. Implemented signals are:

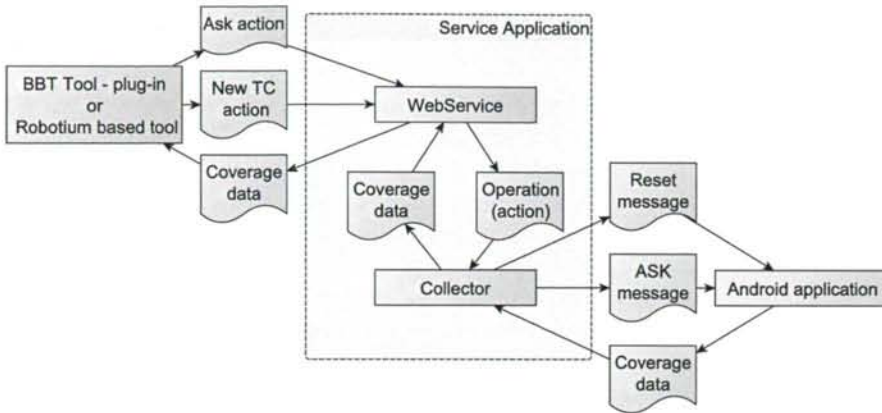


Figure 3: Service Layer

reset With this signal the service asks the apps to reset the stored coverage values.

ask The service sends this signal to query the actual coverage information.

coverage data The application sends this message to the service in response to the **ask** signal. The message contains coverage information.

3.1.3 Installation

To measure coverage on the Android system, two things need to be installed: the particular application we want to test and the common service application that collects coverage information from any instrumented application and provides a communication interface for querying the data from the device.

The service application needs to be installed on a device only once; this single entity can handle the communication of all tested applications.

The instrumented version of each application that is going to be measured must be installed on the Android device. The original version of such an application (if any) must be removed before the instrumented version can be installed. It is necessary because Android identifies the applications by their special android-name and package, and our instrumentation process does not change these attributes of the applications; it only inserts the appropriate instructions into the code. Our toolchain uses the **adb** tool (can be found in Android Development Kit) to remove and install packages.

3.2 Execution

During test execution, the Android device executes the program under test and the service application simultaneously. The program under test counts its own coverage

information and sends this information when the service layer application asks for it. The coverage information can be queried from this service layer application through network connection.

We used two possible modes of test execution: manual and automated. Either mode is used, the service layer application must be started prior to the beginning of the execution of the test cases. It is done automatically by the instrumented applications if the service is not running already.

We implemented a simple query interface in Java for manual testing, a plug-in for the RT-Executor [24], and a simple set of functions for the Robotium [16]. The two automated frameworks use different yet somewhat similar approaches.

On one hand, we used the RT-Executor, which reads the test case scripts and executes the test cases. The client side of the measurement framework is contained in a plug-in of the automation tool, and this plug-in must be controlled from the test case itself. Thus, the test case scripts must be prepared in order to measure the code coverage of the executed applications.

The plug-in can indicate the beginning and the end of the particular test cases to the service, so the service can distinguish the test cases and separate the collected information. In order to measure the test case coverages individually, one instruction must be inserted in the beginning of the test script to reset the coverage values and one instruction must be inserted in the end instructing the plug-in to collect and store coverage information belonging to the test case.

During test execution the following steps are taken:

- The program under test (PUT) is started.
- The start of the program triggers the start of the measurement service if necessary. Then PUT connects to the service and registers itself by its unique name given to it in instrumentation process.
- The test automation system starts a test case. The test case forces the client of the automation system to send a **NEWTC** message to the service. The service sends a **reset** signal to PUT, which resets the coverage array in its collector class. The service returns the actual time to the client.
- The test automation system performs the test steps. PUT collects the coverage data.
- The test case ends. The client of the automation tool sends an **ASK** message to the service. The service sends an **ask** signal to PUT, which sends back the coverage data to the service. The service sends back the coverage data and the actual time to the client.
- The client calculates the necessary information from the coverage data and stores it in the local files. The stored data are: execution time, trace length, coverage value, lists of covered and not covered methods. Another plug-in decides if the test case was passed or failed and stores this information in other local files.

These steps are repeated during the whole test suite execution. At the end, the coverage information of all the executed test cases are stored in local files and are ready to be processed by different stages of the testing methodology.

On the other hand, we used the Robotium framework as a black-box test aiding tool, the Android testing API, and JUnit as the testing environment. Robotium provides useful functions to help accessing the graphical user interface layer of Android applications. This way it makes easy to write JUnit test cases which test any application without user interaction.

In this case, the Android framework executes the JUnit test cases like RT-Executor executes its test scripts. The client-side of the measurement framework is contained in a *TestHelper* class that controls data flow during test execution. Similar to the previous settings, this class must be controlled from the test case itself, so the test cases must also be prepared in order to measure code coverage.

The helper class works like the plug-in of the RT-Executor. Thus, the execution steps are very similar to those mentioned above except that only the coverage information is stored at the end.

3.3 Processing the Data

As we mentioned above, the client side of the coverage measurement system is realized as a plug-in of the RT-Executor tool and as an extension to the Robotium framework.

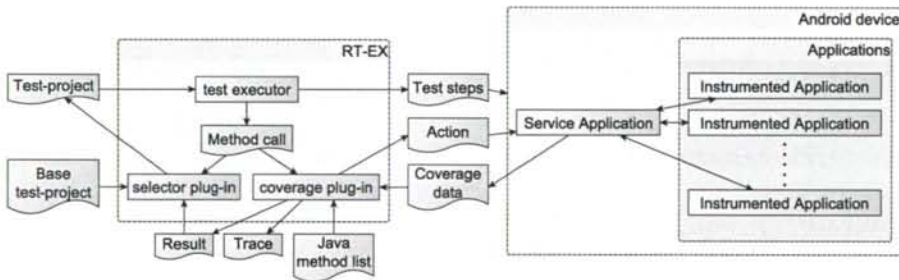


Figure 4: Test execution framework with coverage measurement

In the RT-Executor settings (Figure 4) the plug-in is controlled from the test cases. It indicates the beginning and the end of a test cases to the service layer application. The service replies to these messages by sending the valuable data back. When the measurement client indicates the start of a test case (by sending a **NEWTC** message to the service), the service replies with the current time which is stored by the client. At the end of a test case (when an **ASK** message is sent by the client), the service replies with the current time and the collected coverage information of the methods.

When the coverage data is received, the measurement client computes the execution time, trace length (the number of method calls), and the list of covered and not covered methods' IDs. Then, the client stores these data in a *result* file for further use. The client makes other files, the *trace* files, separately for each test case. Such a trace file stores the identifiers of the methods covered during the execution of the test case.

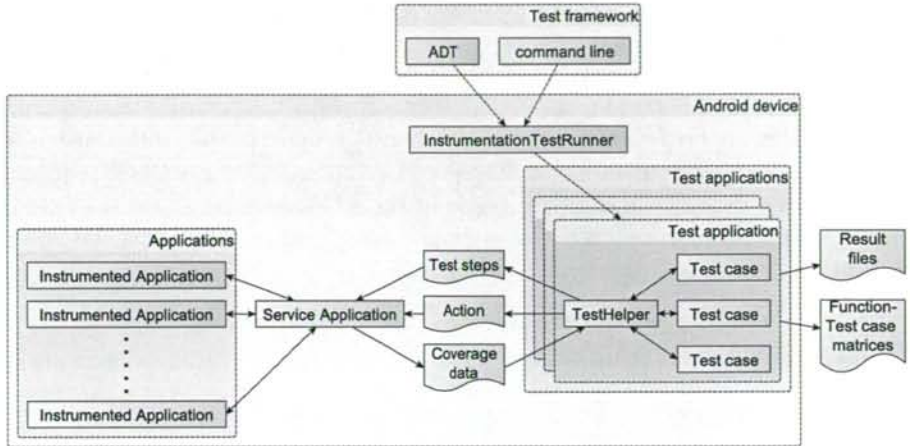


Figure 5: Robotium based test execution environment with the integrated *TestHelper*

In the Robotium settings (Figure 5) the communication between the service layer and the tested application is very similar to the RT-Executor based one. The difference is that the test cases are executed directly by the device and that instead of an external plugin, an internal test helper will communicate with the service application and will produce the coverage data.

As an alternative client, we implemented a simple standalone Java application that is able to connect to the measurement service. This client is able to visualize the code coverage information online, and is useful during the manual testing activities.

3.4 Applications on the Measurement Framework Results

The code coverage and other information collected during the test execution can be used in various ways. In the pilot project, we implemented some of the possible applications. These implementations process the data files locally stored by the client plug-in.

3.4.1 Test Case Selection and Prioritization

Test case selection defines a subset of a test suite based on some properties of the test cases. Test case prioritization is a process that sorts the test suite elements according to their properties [29]. A prioritized list of test cases can be cut at some points resulting in a kind of selection.

Code coverage data can be used for test case selection and prioritization. We implemented some selection and prioritization algorithms as a plug-in of the RT-Executor, which utilizes the code coverage information collected by the measurement framework:

- A change-based selection algorithm that used the list of changed methods and the code coverage information to select the test cases that covered some of the changed methods.
- Two well-known coverage-based prioritization algorithms: one that prefers test cases covering more methods; and another that aims at higher overall method coverage with less test cases.
- A simple prioritization that used the trace length of the test cases.

3.4.2 Not Covered Code

Not covered code plays an important role in program verification. There are two possible reasons for a code part not being covered by any test case executions. The test suite can simply omit its test case, in which case we have to define some new test cases executing the missed code. It can also happen that the not covered code cannot be executed by any test cases, which means that the code is dead. In the latter case, the code can be dropped from the codebase.

In our pilot implementation, automatic test case generation is not implemented. We simply calculate the lists of methods covered and not covered during the tests. These lists can be used by the testers and the developers to examine the methods in question and generate new test cases to cover the methods, or to simply eliminate the methods from the code.

3.4.3 Traceability Calculation

Traceability links between different software development artifacts play a very important role in the change management processes. For example, traceability information can be used to estimate the required resources to perform a specific change or to select the test cases related to the change of the specification. Relationship exists between different types of development artifacts. Some of them can simply be recorded when the artifact is created, some of them must be determined later.

We implemented a traceability calculator that computes the correlation between the requirements and the methods. The correlation computation is based on two binary matrices: the pre-defined relationship matrix between the requirements and

the test cases and the matrix between the test cases and the methods (code coverage). From these matrices a binary vector can be assigned to each requirement and method representing whether the test cases assigned to the elements of this vector have relationship to the given requirement or method. If a requirement-method pair is assigned with high correlation (i.e., their assigned binary test case vectors are highly correlated), we can assume that the required functionality is implemented in the method. To calculate the correlation of these binary vectors we implemented three different well-known methods: the Pearson's product-moment coefficient [25], the Kendall's correlation coefficient [25], and a Manhattan distance based method where the similarity coefficient was defined as

$$S_M(a, b) = \frac{1}{1 + \sum_{i=1}^n |a_i - b_i|}. \quad (1)$$

The use of the information that is extracted from the results of the correlation computational processes can be diverse. For example, it can be used to assess the number of methods to be changed if the particular requirement changes. Additionally, as we observed during our usability testing project, if we define functionalities closely related to the usage of UI elements, then it can indicate the relations between these graphical elements and the parts of the code-behind.

4 Usage and Evaluation

In this section, we present and evaluate some use cases to demonstrate the usability of the measurement toolchain.

4.1 Additional Test Case Generation

In the pilot project our target embedded hardware was an Android-based Set-Top-Box. We had this device with different pre-installed applications and test cases for some of these apps. Considering the available resources we decided to test our methodology and implementation on a media settings application. After executing the tests of this application with coverage measurement, we found that the pre-defined tests covered only 54% of the methods. We examined the methods and defined new test cases. Although the source code of this application was not available, based on the not covered method names and the GUI, we were able to define new test cases that raised the proportion of covered methods to 69%. This is still far from the required 100% method level coverage, but shows that the feedback on code coverage can be used to improve the quality of the test suite.

4.2 Traceability Calculation

We made two experiments with the framework using it for traceability calculation.

First, in the CIRENE pilot project a simple implementation that is able to determine the correlation between the code segments and the requirements was made.

We did not conduct detailed experimentation in this topic, but we did test the tool. Instead of the requirements, we defined 12 functionalities performed by three media applications (players) on our target Set-Top-Box device. Then, we assigned these functionalities to 15 complex black-box test cases of the media applications and executed the test cases with coverage measurement. The traceability tool computed correlations between the 12 functionalities and 608 methods, and was able to separate the methods relevant in implementing a functionality from the not relevant methods.

In the experiment connected to the usability testing project our direct goal was to investigate whether the coverage information could be used to determine a small set of program locations to be instrumented in order to collect relevant information for usability analysis. The main idea was that by reducing the number of instrumentation points needed for comprehensive usability testing we would be able to minimize the possible negative effects on the performance of the application under test and, therefore, analysing complex applications would become easier. We conducted an experiment involving 10 small to medium sized Android applications (see Table 1). Test cases were created for the applications each one modelling some typical complex usage sessions. Next, we defined some functionalities for each application. This measurement aimed to verify that automatic methods are able to uncover relevant traceability links, and to evaluate the efficiency of different correlation computation methods in indicating traceability links between the artefacts.

Table 1: List of applications used for experiments

Application	Classes	Methods	Functionalities	Test cases
A_0	134	671	4	13
A_1	144	1083	4	25
A_2	303	1675	5	12
A_3	545	2565	9	12
A_4	812	3897	5	14
A_5	861	6760	5	11
A_6	1257	9619	5	12
A_7	1519	11854	5	11
A_8	1537	11166	5	15
A_9	4247	24747	5	12

In order to evaluate the results of the three different computations we examined the functionalities and methods of the applications and created *reference* links between them by manually classifying the methods of each application and connecting them to the functionalities. First the functionalities were determined by usage scenarios. Next, we used some kind of semantic similarity: words semantically connected to the determined functionalities and usual Android UI element name fragments were searched for in the class and method names. Functionalities

were initially assigned with the matching elements. Then this initial classification was refined manually by examining each program element and looking for hidden or false *reference* links.

For evaluating the traceability calculation methods and comparing them to our manual method, we used the *precision*, *recall*, and *F-measure* metrics [23]. The first step of assessing these metrics was to compare the manually determined *reference* links to the function-method traceability links that were selected by the different correlation based traceability calculation methods. The comparison of the *reference* and the computed links classified each traceability link as true or false positive, and each lack of link as true or false negative records [23] for a calculation method. Based on this classification of links, the three metric values were computed for each traceability calculation method and for all applications.

All of the used correlation computation methods assigns a real value within an interval $([-1, 1]$ or $[0, 1])$ to a functionality-method pair, but the existence of the link is a binary information. To evaluate the methods we had to define some thresholds to convert real values into true and false values. As the different methods give different numbers, we could not use the same value for all the three ones. Thus, we checked the precision, recall and F-measure values of different threshold values for each methods and computed averages for all applications. The results are shown in Figure 6.

By comparing the curves, we can observe that precision first slightly improves as the threshold grows, then it suddenly drops. Although completeness cannot be totally ignored, for our purposes less noise (fewer false positives) in the generated data is more important than completeness. Thus, we have chosen to select thresholds where the precision is maximal before its drop down. It resulted in 0.8, 0.3, and 0.1 threshold values for methods Pearson, Manhattan, and Kendall, respectively.

Table 2 shows the precision, recall, F-measure values of the three computation methods using the previously defined threshold values. As can be seen, in half of the cases the Pearson method produces the smallest set, and in four cases of them this is the best choice according to the precision. Manhattan and Kendall methods produce the same smallest sets in three cases and each of them produce the smallest set individually in one case. However, the precision for these sets is always the best among the three methods.

These results show that any of these three methods can be effectively used for calculating traceability between source code and functionalities of a software. For these 10 applications, the Pearson method seems to be slightly better than the other two, but the results are not convincing. Which is the best is probably depending on some other characteristics of the software.

Based on these results we can say that the investigated methods that infer traceability links from code coverage data can be used to identify program points whose inspection provide relevant information for usability testing. The effectiveness of these methods are comparable to the manual traceability link detection. Therefore, it is possible to use them to support the usability testing of large sized Android applications.

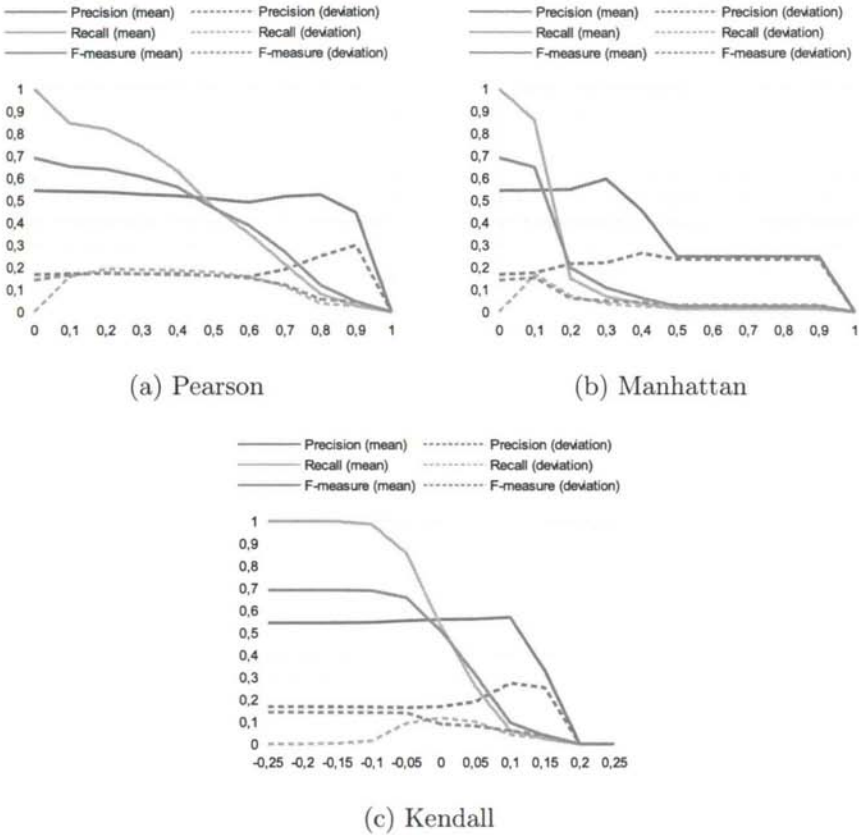


Figure 6: Precision, recall, and F-measure values at different thresholds for the three methods. (X axis: treshold; Y axis: metric value.)

5 Conclusions and Future Work

In this paper, we presented a methodology for method level code coverage measurement on Android-based embedded systems. Although there were more solutions allowing the measure of the code coverage of Android applications on the developers' computers, no common methods were known to us that performed coverage measurement on the devices. We also reported the implementation of this methodology on a digital Set-Top-Box running Android. The coverage measurement was integrated in the test automation process of this device allowing the use of the collected coverage data in different applications like test case selection and prioritization of the automated tests, or additional test case generation. We also presented an application of the framework. Using the produced coverage data we performed experiments with three traceability computation methods.

Table 2: Precision (P), recall (R), F-measure (F) values for applications and computation methods

Application	(a) Pearson 0.8			(b) Manhattan 0.3			(c) Kendall 0.1		
	P	R	F	P	R	F	P	R	F
A ₀	0.78	0.13	0.22	0.79	0.10	0.17	0.79	0.10	0.17
A ₁	0.27	0.06	0.10	0.27	0.08	0.12	0.00	0.00	0.00
A ₂	0.35	0.07	0.10	0.76	0.04	0.07	0.76	0.04	0.07
A ₃	0.07	0.01	0.02	0.28	0.13	0.16	0.28	0.13	0.16
A ₄	0.54	0.07	0.12	0.46	0.02	0.04	0.46	0.02	0.04
A ₅	0.63	0.05	0.06	0.54	0.03	0.05	0.54	0.03	0.05
A ₆	0.81	0.09	0.14	0.86	0.10	0.16	0.86	0.10	0.16
A ₇	0.44	0.13	0.12	0.66	0.09	0.11	0.66	0.09	0.11
A ₈	0.83	0.10	0.17	0.84	0.08	0.14	0.84	0.08	0.14
A ₉	0.52	0.08	0.19	0.50	0.03	0.05	0.50	0.03	0.05
Average	0.52	0.08	0.12	0.60	0.07	0.11	0.57	0.06	0.10
Deviation	0.25	0.04	0.06	0.22	0.04	0.05	0.27	0.04	0.06

There are many improvement possibilities of this work. Regarding the implementation of code coverage measurement on Android devices, we wish to examine if the granularity of tracing could be fined to sub-method level (e.g., to basic block or instruction levels) without significantly affecting the runtime behaviour of the applications. This would allow us to extract instruction and branch level coverages that would result in more reliable tests. In addition, we are thinking of improving the instrumentation in order to build dynamic call trees for further use. The current implementation (simple coverage measurement) does not deal with timing, threads and exception handling, which are necessary for building the more detailed call trees. It would also be interesting to help the integration of this coverage measurement in commonly used continuous integration and test execution tools.

Furthermore, we are examining the use of the resulting coverage data. There are other ways code coverage and computed traceability information can be used in usability testing, for example to partially automate collecting data and to establish usability models. The implemented code coverage measurement and the testing process that utilizes this information are a good base for measuring the effect of using coverage measurement data on the efficiency and reliability of testing.

References

- [1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, Oct 2002.
- [2] Apache Commons. BCEL homepage. <http://commons.apache.org/proper/commons-bcel/>, June 2013.
- [3] Beszédes, Árpád, Gergely, Tamás, Papp, István, Marinković, Vladimir, and Zlokolica, Vladimir. Survey on testing embedded systems. Technical report, Department of Software Engineering, University of Szeged, and Faculty of Technical Sciences, University of Novi Sad, 2012.

- [4] Biswas, Swarnendu, Mall, Rajib, Satpathy, Manoranjan, and Sukumaran, Srihari. A model-based regression test selection approach for embedded applications. *SIGSOFT Softw. Eng. Notes*, 34(4):1–9, July 2009.
- [5] Bornstein, Dan. Presentation of Dalvik VM internals, 2008.
- [6] Chawla, Anil and Orso, Alessandro. A generic instrumentation framework for collecting dynamic information. In *Online Proc. of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, 2004.
- [7] Chiba, Shigeru. Javassist homepage. <http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/>, May 2013.
- [8] Cleland-Huang, Jane, Czauderna, Adam, Gibiec, Marek, and Emenecker, John. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 155–164, New York, NY, USA, 2010. ACM.
- [9] Costa, José C., Devadas, Srinivas, and Monteiro, José C. Observability analysis of embedded software for coverage-directed validation. In *Proceedings of the International Conference on Computer Aided Design*, pages 27–32, 2000.
- [10] Developers. JSON. <http://www.json.org/>, June 2013.
- [11] Eaddy, M., Aho, A.V., Antoniol, G., and Gueheneuc, Y.-G. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 53–62, June 2008.
- [12] Google. Android homepage. <https://www.android.com/>, June 2013.
- [13] Google. apktool homepage. <https://code.google.com/p/android-apktool/>, May 2013.
- [14] Google. dex2jar. <https://code.google.com/p/dex2jar/>, May 2013.
- [15] Google Android Developers. Building and running an android application. <http://developer.android.com/tools/building/index.html>, May 2013.
- [16] Google Code. Robotium homepage. <https://code.google.com/p/robotium/>, March 2014.
- [17] Gotlieb, Arnaud and Petit, Matthieu. Path-oriented random testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 28–35, New York, NY, USA, 2006. ACM.
- [18] Hazelwood, Kim and Klauser, Artur. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

- [19] Kukolj, Sandra, Marinković, Vladimir, Popović, Miroslav, and Bognár, Szabolcs. Selection and prioritization of test cases by combining white-box and black-box testing methods. In *Proceedings of the 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2013)*, 2013.
- [20] Marcus, A. and Maletic, J.I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135, May 2003.
- [21] Marek, Lukáš, Zheng, Yudi, Ansaloni, Danilo, Sarimbekov, Aibek, Binder, Walter, Tůma, Petr, and Qi, Zhengwei. Java bytecode instrumentation made easy: The disl framework for dynamic program analysis. In Jhala, Ranjit and Igarashi, Atsushi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer Berlin Heidelberg, 2012.
- [22] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions on*, 33(6):420–432, June 2007.
- [23] Powers, David MW. Evaluation: from precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.
- [24] RT-RK Institute. RT-Executor. <http://bbt.rt-rk.com/software/rt-executor/>, May 2013.
- [25] Salkind, N.J. *Encyclopedia of measurement and statistics*. Number 1. k. in *Encyclopedia of Measurement and Statistics*. SAGE Publications, 2007.
- [26] Seesing, A. and Orso, A. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the Eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, San Diego, CA, USA, october 2005.
- [27] Slife, Derek and Chesney, Mark. jCello. <http://jcello.sourceforge.net/>, June 2013.
- [28] Vlad Roubtsov. EMMA: a free java code coverage tool. <http://emma.sourceforge.net/>, June 2013.
- [29] Yoo, S. and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

REGULAR PAPERS

Hungarian Noun Phrase Extraction Using Rule-based and Hybrid Methods*

Gábor Recski[†]

Abstract

We implement and revise Kornai's grammar of Hungarian NPs [11] to create a parser that identifies noun phrases in Hungarian text. After making several practical amendments to our morphological annotation system of choice, we proceed to formulate rules to account for some specific phenomena of the Hungarian language not covered by the original rule system. Although the performance of the final parser is still inferior to state-of-the-art machine learning methods, we use its output successfully to improve the performance of one such system.

Keywords: natural language processing, parsing, machine learning

1 Introduction

This paper describes a rule-based system which extracts noun phrases (NPs) from morphologically analyzed Hungarian text. We implement and revise the grammar of Hungarian NPs in [11] to create a system that identifies NPs by means of bottom-up parsing. Although high performance on the standard task is already possible using state-of-the-art machine learning methods, we show that a rule-based approach contributes substantially to the performance of a hybrid system. Section 2 describes the task and provides a brief survey of the standard statistical approach. Section 3 documents the process of creating a Hungarian NP corpus, a resource crucial not only for machine learning approaches, but also for the evaluation of rule-based systems.

Section 4.1 describes the technical preliminaries of creating an NP parser. In section 4.2 we describe the process of grammar development, which involves examining the error classes in the output after every major change to the rule system. Section 5 proposes a simple hybrid system where the chunking task is performed by the learning-based system *hunchunk* [23], [24] using features derived from the output of the rule-based system.

*I would like to thank András Kornai and two anonymous reviewers for their many useful comments and suggestions. Work supported by OTKA grant #82333.

[†]Department of Theoretical Linguistics, Eötvös Loránd University and Hungarian Academy of Sciences, E-mail: recski@budling.hu

2 Chunking

2.1 The task

The task of extracting one or several types of phrases from a text is often referred to as *shallow parsing* or *chunking*. The term *chunk* and the task of text chunking, however, do not have universally accepted definitions in NLP (Natural Language Processing) literature. The term *chunk* was first used by Abney in [2], who uses it to describe non-overlapping units of a sentence that each consist of “a single content word surrounded by a constellation of function words”. Based primarily on [9], who introduce the term *performance structure* to describe psycholinguistic units of a sentence, Abney argues that chunks are units that do not necessarily coincide with syntactic constituents. Recent works on the automated chunking of raw text, however, invariably use definitions of chunks that make it possible to extract them from parse trees in order to provide training data for supervised learning systems. In practice, these chunks usually coincide with some group of syntactic phrases. One complete set of definitions for various classes of chunks is given in the description of the chunking task of CoNLL 2000 [28], where the Penn Treebank [17] was used as a source of chunk data.

One of the best known works on the extraction of NP chunks is that of Ramshaw and Marcus [18], who define *base NPs* (or *non-recursive NPs*) as noun phrases that do not contain another noun phrase. It is this definition that was adopted by Tjong Kim Sang and Buchholz for the CoNLL 2000 shared task, and when the task of NP chunking is mentioned as a benchmark for some machine learning algorithm, it almost invariably refers to base NP tagging based on the datasets proposed by Ramshaw and Marcus and adopted by CoNLL-2000.

2.2 Overview of statistical methods

Besides defining the task of NP chunking as the identification of non-recursive (base) noun phrases, Ramshaw and Marcus attempt to solve the task by applying the method of transformation-based learning, which had been used before for the tasks of part-of-speech tagging [4] and parsing [5]. Using the datasets and method of evaluation that was later to become the CoNLL shared task and also the standard field of comparison for NP-chunker tools, Ramshaw and Marcus report precision and recall rates of 90.5% and 90.7% respectively. Their datasets used for training and testing purposes were derived from sections 15-18 and section 20 of the Wall Street Journal respectively, data which was available from the Penn Treebank.

During and after the CoNLL shared task in 2000, a wide variety of machine learning methods have been applied to the task of identifying base NPs. Kudo and Matsumoto reached an F-score of 93.79% by using Support Vector Machines [13], a result that was to increase to 94.22% a year later when they introduced weighted voting between SVMs trained using different chunk representations [14]. Probably the most popular method for NP chunking today is the Conditional Random Field (CRF, [15]) machine learning algorithm. CRFs have been used on the standard

CoNLL task by Sha and Pereira, who achieved an F-score of 94.3% [26], and more recently by Sun et al. ($F = 94.34\%$) [27].

A further notable result is that of Hollingshead and colleagues [10], who evaluated several context-free parsers on various shallow parsing tasks and report an F-score of 94.21% on the CoNLL task using the Charniak parser [6]. These results show that a rule-based system can be competitive with results obtained by using any advanced machine learning algorithm, a fact that clearly points us in the direction of hybrid systems.

2.3 The hunchunk system

In the final section of this paper we shall combine the parser with our own learning-based NP-chunking tool. **Hunchunk** uses a combination of Maximum Entropy learning and Hidden Markov Models (HMM) to perform NP-chunking of a sentence that is tokenized and morphologically annotated. For a detailed description of **hunchunk** the reader is referred to [23]. Some past applications of **hunchunk** are documented in [25] and [22]. The tool is available for download under an LGPL license from <http://www.github.com/recski/HunTag>.

3 Creating NP corpora

A preliminary step of creating the NP corpus is choosing a method for representing morphological information. The morphological analyzer **hunmorph** [29] uses the KR formalism [20] and our grammar relies heavily on the kind of structured information that **hunmorph** provides and KR codes represent.

3.1 The KR formalism

The KR formalism for representing morphological information was developed with the intention of capturing the hierarchy between individual inflectional features and encoding the derivational steps used to arrive at the word form in question. The output of the analysis of a word starts with the stem and contains the category and features of the word as well as the category of the word from which the given form was derived, if any. This latter part of the code also contains in square brackets the type of derivation used to form the final word. The last part of the code represents the hierarchy between grammatical features of the word by means of bracketing similar to that used for the analysis of sentence structure.

Some examples of KR-codes in the Szeged Treebank [7] are given in Table 1. As can be seen, KR encodes the entire chain of derivations that led to the word form under analysis.

One great advantage of this formalism is that it explicitly encodes all pieces of information which one might think of as a grammatical feature, therefore any NLP application which relies on word level information can make use of the KR code

Table 1: KR examples

<i>tanárunk</i>
teacher-Poss1Pl
'our teacher'
<i>tanár</i> /NOUN<POSS<1><PLUR>>
<i>óráján</i>
class-Poss3-SUP
'in his/her class'
<i>óra</i> /NOUN<POSS><CAS<SUE>>
<i>másodikkal</i>
two-ORD-INS
'with the second'
<i>kettő</i> /NUM[ORD] /NUM<CAS<INS>>
<i>vegyük</i>
take-Imp-Pl1-Def
'let's take'
<i>vesz</i> /VERB<SUBJUNC-IMP><PERS<1>><PLUR><DEF>
<i>felértékelődése</i>
up-value(V)-Med-Ger-Poss3
'the increase of its value'
<i>felértékel</i> /VERB[MEDIAL] /VERB[GERUND] /NOUN<POSS>

without the need for any external knowledge about the meaning of various symbols or positions in the code.

The KR formalism straightforwardly encodes most grammatical features, but there are still some distinctions which it is unable to represent. One of these, which we must overcome in order to account for syntactic phenomena, is the distinction between pronouns and nouns as well as the various types of pronouns in Hungarian. Pronouns are tagged as nouns in the KR formalism because they take part in the same inflectional phenomena as nouns – although some of their paradigms are defective –, therefore introducing a new top-level category into the KR system would cause the loss of a well-founded generalization. The solution we implemented for use with our system is the introduction of the noun feature PRON which takes as its value 0 if the word is not a pronoun and the type of pronoun otherwise. This addition results in the analyses exemplified in Table 2, for a detailed description see [21].

Table 2: Pronoun types

<i>ez</i>
this
ez/NOUN<PRON<DEM>>
<i>mindenki</i>
everybody
mindenki/NOUN<PRON<GEN>>
<i>valami</i>
something
valami/NOUN<PRON<INDEF>>
<i>aki</i>
who (relative pron.)
aki/NOUN<PRON<REL>>
<i>ki</i>
who (interrogative pron.)
aki/NOUN<PRON<WH>>
<i>saját</i>
own
aki/NOUN<PRON<POS>>

3.2 Extracting NPs from a treebank

Having determined the way we wish to encode morphological information we may proceed to create an NP corpus by extracting sentences and syntactic information from a treebank (a corpus which contains the full syntactic analysis for all sentences, cf. [1]). For this purpose we use the Szeged Treebank [7], a syntactically annotated corpus of Hungarian which contains nearly 1.5 M tokens of text taken from a variety of genres including fiction, newspapers, legal text, software documentation and essays written by students between the age of 13 and 16.

The treebank contains morphological information about each word in the MSD format [8]. Converting MSD-tags to KR is insufficient because MSD codes do not contain data about the derivations that create a word form, a piece of information which KR can encode and which some of our rules rely on. Our morphological analyzer, *hunmorph*, is able to supply this information, but it will necessarily produce some sporadic tagging errors on sentences extracted from the Treebank. Such errors may be corrected in a machine learning system based on context, but will surely mislead the rule-based system, which has no other source of information at

its disposal. In order to have all available data present in the corpus, and at the same time preserve the high precision provided by manually annotated tags, we merged our two sources of data. Information on the derivation of a word form, if any, was taken from the KR-codes provided by *hunmorph*, the remaining part of the tag, containing the category of the word as well as its grammatical features, was obtained from the Treebank. In case the Treebank could not provide any grammatical information (0.91% of all words), the output of *hunmorph* was entered into the corpus as is.

3.3 Mending the corpus

Having created a base NP corpus by the method described in section 3.2, we proceeded to apply two further changes to the data in order to handle syntactic analyses in the Treebank with which we do not agree. Since we intend to use these corpora as a standard of evaluation for the parser, we need it to reflect the analyses which we expect our system to produce. In this paper we do not wish to argue extensively for one analysis over the other, we simply describe the changes we have made to the data in order to ensure that our experiments can be replicated.

3.3.1 Adjectives in possessive constructions

The largest number of cases where there is a discrepancy between the Szeged analysis and the one used here is related to the analysis of possessive constructions. The noun phrase in Table 3 is represented in the treebank as in Figure 1.

Table 3: Possessive construction

<i>egy</i>	<i>idős</i>	<i>úr</i>	<i>kopasz</i>	<i>fejére</i>
an	elderly	gentleman	bald	head-Poss3-SUBL
'on the bald head of an elderly gentleman'				



Figure 1: Original analysis of the possessive construction

We believe this analysis to be false since the noun and preceding adjective modifying it form a constituent in Hungarian and the possessive construction does not change this fact: the possessor NP can be followed directly by any NP with the POSS feature. Therefore we modified our base NP corpus in order to reflect the

analysis in Figure 2, which we believe to be the correct one. We will expect our system to parse such structures as two consecutive NPs.

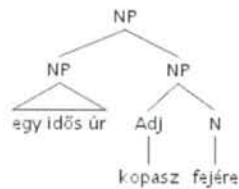


Figure 2: Revised analysis of the possessive construction

3.3.2 Demonstratives

Another structure which we intend to treat differently from the analysis in the Treebank is the special demonstrative construction of Hungarian exemplified in Table 4. Note that in this structure the demonstrative pronoun *ez/az* must be marked for both the case and number of the following noun.

Table 4: Demonstrative NP

<i>ez</i>	<i>a</i>	<i>pincér</i>
this	the	waiter
'this waiter'		
<i>ezek</i>	<i>a</i>	<i>hajók</i>
this-PL	the	ship-PL
'these ships'		
<i>attól</i>	<i>a</i>	<i>pasastól</i>
that-ABL	the	bloke-ABL
'from that bloke'		

For these structures the Treebank gives the analysis in Figure 3. We believe that the demonstrative pronoun cannot project a noun phrase of its own, therefore we change the corpus to reflect the analysis in Figure 4.

3.3.3 Other issues

The chunk corpus extracted from the Szeged Treebank still present a number of small anomalies that hinder the evaluation of both the rule-based and the statistical system as well as the training of the latter. One notable example is a construction which involves an NP containing an adjective that precedes the noun and is enclosed

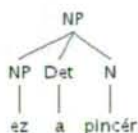


Figure 3: Original analysis of demonstrative NPs

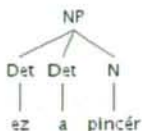


Figure 4: Revised analysis of demonstrative NPs

in parentheses and which occurs often in legal text (e.g. *A Gt. (új) 3. paragrafusa* ‘The (new) 3rd section of the Gt. Act’). This case falls under the same questionable analysis as those described in section 3.3.1. We believe that arbitrary modification of the analysis of problematic structures (which are, unfortunately, overrepresented in our corpus) is not a measure we can take in good conscience. Therefore, we leave these occurrences, as well as any smaller anomalies, untouched. We note that this phenomenon accounts for ca. 5% of those base NPs which our grammar is unable to parse.

3.4 Evaluation methods

The corpus created in the manner described above is used to evaluate our parser at various stages of development. The statistical system *hunchunk* also uses two (non-overlapping) sections of this corpus for training and testing. Finally, performance of the hybrid system (to be introduced in the final section of this paper) is also measured using this data as gold standard.

In each of these cases, evaluation involves comparing two sets of chunks for each sentence, the one supplied by the system in question and the one present in the corpus. Our evaluation method follows the guidelines of CoNLL-2000: a chunk identified by our system is considered correct iff it corresponds to a chunk in the gold standard and a chunk in the corpus is considered found iff it corresponds to a chunk in our tagging. A system’s performance can be described by two values: the *precision* of a system is the number of correctly identified chunks divided by the number of all chunks in the output, while the *recall* rate is obtained by dividing the same number by the number of chunks in the gold standard. As customary, we measure the overall quality of the tagging by calculating the harmonic mean of these two values, also called the F-score:

$$F = \frac{2PR}{P + R}$$

where P and R stand for precision and recall respectively (cf. e.g. [16]).

4 Rule-based method

This chapter describes our efforts to use a rule-based parser for the extraction of noun phrases. We improve the context-free feature grammar of Hungarian NPs [11] [12] in order to account for even the most complicated structures.

4.1 Building a parser

Our system uses the NLTK parser, a tool which supports context-free grammars and a wide variety of parsing methods [3]. To parse a text we must first give a feature representation of all words. We implement the context-free grammar of Kornai to create a parser which takes as its input the series of KR-codes of words in a sentence and produces, by means of bottom-up parsing, charts containing the possible rule applications that may produce some fragment of the sentence. A chunking is then derived from this chart through a series of recognition steps which we shall describe at the end of this section.

4.1.1 Preparing the data

When using the NLTK parser with a CF grammar, the system accepts nonterminal symbols that consist of a category symbol such as NOUN or VERB followed by a set of features in square brackets. Feature values can be strings, integers, non-terminals of the grammar and variables that bind the value of the feature to that of some other feature of the same type in the rule. Thus a rule to encode agreement in number between verb and object would be $VP \rightarrow V[PL=?a] N[PL=?a]$, which is equivalent to the more standard ‘Greek variable’ notation $VP \rightarrow V[\alpha PL] N[\alpha PL]$. Converting KR codes to such representations, i.e. supplying the terminal rules for our grammar, is a straightforward mechanical process. Some examples are given in Table 5. Notice that the grammar does not use different symbols for various projection levels of the same syntactic category, but encodes this information in the feature BAR; the notation NOUN[BAR=0] will then simply represent a bare noun. Information on the source of derivation is represented by the feature SRC which takes as its value a set of two features: STEM encoding the features of the source word and DERIV the type of derivation.

As we have described in section 3.1, the bulk of any KR-style code lends itself to such a representation, e.g. the code NOUN<POSS><PLUR> needs only to be rewritten as NOUN[POSS=1, PLUR=1] in order to produce input for NLTK. Still, a number of problems must be addressed when transforming KR codes into such feature structures. First of all, KR features are privative: the fact that a noun is singular, for example, can be concluded from the absence of the <PLUR> feature. Similarly, the default case is nominative (there is no <CAS<NOM>> feature), the default person is the third, etc. Since our grammar should be able to refer to such default features

Table 5: Terminal rules

NOUN[POSS=[1=1, PLUR=1] -> NOUN<POSS<1><PLUR>>
NOUN[POSS=1, CAS=[SUE=1]] -> NOUN<POSS><CAS<SUE>>
NOUN[ANP=0, CAS=0, PLUR=0, POSS=[1, PLUR=1], PRON=0] -> -> 'NOUN<POSS<1><PLUR>>
NUM[CAS=[INS=1], SRC=[STEM=NUM, DERIV=ORD]] -> -> NUM[ORD]/NUM<CAS<INS>>
VERB[SUBJUNC-IMP=1, PERS=[1=1], PL=1, D=1] -> -> VERB<SUBJUNC-IMP><PERS<1>><PLUR><DEF>
NOUN[POSS=1, SRC=[STEM=VERB[SRC=[STEM=VERB, DERIV=MEDIAL]], DERIV=GERUND]] -> -> VERB[MEDIAL]/VERB[GERUND]/NOUN<POSS>

in a straightforward manner, the process of transforming KR-codes involves explicating these features by adding the feature values PERS=0, CAS=0, PLUR=0, etc. Similarly, a word which has not been identified as the product of some derivation will receive the feature SRC=0.

4.1.2 Implementing NP-chunking

Having established a method for creating the terminal rules of our grammar we are now able to parse, based on the NP-grammar of Kornai, any sentence tagged according to the KR formalism. Since we do not have a complete grammar of Hungarian, we employed a bottom-up parser, which can provide an analysis of fragments of a sentence without parsing the full sentence. The output obtained for each sentence is a chart which contains *edges*, individual entries which describe a step in the parsing process by representing a particular application of a rule in the grammar, and gives the location of the sentence fragment to which it can be applied.

The absence of an S-grammar means that we cannot automatically discard the majority of chart edges based on their lack of ability to function as part of a parse-tree for the full sentence. Therefore we must compile a list of rules to post-process the set of parse edges in order to produce non-overlapping NP sequences. First, we take all fragments of the sentence which correspond to a *complete* NOUN edge, thereby selecting the word sequences that the parser considers potential NPs of the sentence. Secondly, since we are trying to extract base NPs only, we discard all fragments which contain more than one noun. Next, we discard all fragments

which are contained in a larger fragment. The final and most complicated step in finding NPs is dealing with overlapping fragments: we implement a heuristic approach in which we choose of two overlapping NPs the one which cannot be parsed as a phrase of some other category based on the parse chart. This process is preferable since most overlaps are produced by SLASH-rules, i.e. rules which allow NPs with elliptic heads to be parsed as NPs. In most cases, these rules falsely generate phrases which are not NPs but AdjPs, NumPs, etc. In case this process fails to select exactly one of the two fragments – i.e. both or neither of them can be parsed as a phrase of some other category – we discard them both.

4.2 Developing the grammar

In this section we describe our additions to the grammar of Hungarian NPs published in [11]. We evaluate each version of the grammar on a test corpus which contains 1000 sentences picked randomly from all genres in the base NP corpus, following the principles described in section 3.4.

Implementing the initial grammar of Kornai our system achieves an F-score of 81.76%. By observing the output it is clear that the greatest shortcoming of our system is its lack of knowledge about the internal structure of adjectival, numeral and adverbial phrases, all of which can form components of an NP. Therefore our first step does not involve touching the NP grammar but rather the addition of some simple rules to account for complex AdjPs, NumPs and AdvPs. These rules can be seen in Table 6.

Table 6: Basic rules for AdjPs and NumPs

ADJ	->	ADJ	ADJ
ADJ	->	ADV	ADJ
NUM	->	NUM	NUM
NUM	->	ADV	NUM
NUM	->	ADJ	NUM

After the addition of these rules our system produces chunkings with an F-score of 84.18%. The next step involved the treatment of pronouns. We have discussed in section 3.1 that Hungarian pronouns behave very similarly to nouns, and in fact the parser can only distinguish them from nouns with the help of a feature which we have added to the KR-system. In the vast majority of cases, treating pronouns as nouns is entirely justified. There are, however, a handful of phenomena which make it necessary for us to refer to them separately in the grammar. General pronouns (e.g. *minden* 'all') and indefinite pronouns (e.g. *néhány* 'some') may combine with a following noun constituent to form an NP (cf. Table 7)

These pronouns are not in complementary distribution with numerals, however we choose to keep the grammar simple and adjoin them to nouns of bar-level 1. The resulting rules are shown in Table 8.

Table 7: General and indefinite pronouns

<i>minden</i>	<i>pofon</i>
all	punch
'all punches'	
<i>néhány</i>	<i>villanykörte</i>
some	light-bulb
'some light-bulbs'	

Table 8: Rules for general and indefinite pronouns

```

NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
-> NOUN[PRON=GEN]
NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]

NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
-> NOUN[PRON=INDEF]
NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]

```

The addition of these two rules result in an increase of the system's F-score to 85.45. A third type of pronoun, the demonstrative *ez/az*, etc. also needs treatment when it comes to the demonstrative structure described in section 3.3.2. To allow the parser to recognize the structure we implement the rule in Table 9, thus achieving an F-score of 86.68.

Table 9: Rule for demonstrative NPs

```

NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e] ->
-> NOUN[PRON=DEM, BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d]
ART NOUN[PRON=0, BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=0],

```

The next structure which caused numerous parsing errors is that of adjectival phrases containing a noun followed by an adjective derived from a verb (called a *deverbal adjective*), either in perfect or imperfect participle form. An example of both of these structures can be seen in Table 10.

Since our terminal symbols encode information about the source of derivation which produced any given word form, we can once again treat these structures properly by adding the two rules in Table 11 to our grammar. This addition caused an increase in the performance of the system to 87.87%. In the end the greatest

Table 10: Sentences with deverbal adjectives

<i>a</i>	<i>korsónak</i>	<i>támasztott</i>
the	jug-DAT	prop-PERF_PART
<i>könyvet</i>	<i>olvasta</i>	
book-ACC	read-PAST-DEF-3	
'He read the book propped up against the jug.'		
<i>az</i>	<i>ókori</i>	<i>mór</i>
the	ancient	moor
<i>hódítóktól</i>	<i>származó</i>	<i>esküvést</i>
conqueror-Pl-FROM	originate-IMPERF_PART	oath-ACC
<i>hallották</i>		
hear-PAST-DEF-3		
'They heard the oath originating from ancient moor conquerors'		

Table 11: Rules for deverbal adjectives

ADJ -> NOUN ADJ [SRC=[STEM=VERB [], DERIV='PERF_PART']]
ADJ -> NOUN ADJ [SRC=[STEM=VERB [], DERIV='IMPERF_PART']]

error classes – besides those caused by genuinely ambiguous structures – remained those which involved the incorrect parsing of punctuation marks and conjunctions. With the addition of several rules describing their behaviour in and around NPs (see Appendix A) we further increased the F-score of the system to 89.36%.

The progress of the system’s performance as a result of our steps of grammar development are summarized in Table 12. As can be seen from these figures our

Table 12: Stages of grammar development

Development stage	F-score
Kornai 1985	81.76%
AdjPs, AdvPs, NumPs	84.18%
Pronouns	85.45%
Demonstrative NPs	86.68 %
Deverbal adjectives	87.87%
Punctuation and conjunctions	89.36%

development of the grammar corrected nearly half of the errors made by the system. For the final version of the grammar see Appendix A.

5 Features from the parser

Although the performance of our parser is still inferior to statistical systems, in this final section we will demonstrate, using a very simple example, how a machine learning system may benefit from the output provided by the parser.

Hunchunk handles the task of chunking as a type of word-tagging and attempts to assign the correct chunk-tag to each word in the sentence: the five tags **B-NP**, **I-NP**, **E-NP**, **1-NP**, **0** indicate the position of a word within a chunk and each possible chunking of a sentence corresponds to a sequence of word tags. The system uses Maximum Entropy learning [19] to determine for each word the probability distribution over this tagset, based on a set of binary features of the word such as character ngrams, morphological features, position in the sentence, etc. (see [23] for details). Using these distributions as observation probabilities and a simple bigram model as an estimate for transition probabilities, the Viterbi algorithm can efficiently compute the most probable sequence of tags, i.e. the most probable chunking for a sentence.

We improve the system by first converting the output of the NP-parser to such a sequence of tags and then using the tag for each word as an extra feature that the maxent model has access to. In other words, when trying to guess what the chunk-tag of a word should be, the **hunchunk** system may use the answer the NP-parser gives to the same question. In order to evaluate this hybrid system we parse the entire chunk corpus and then create a train and test set from the data obtained in the same way as we would do when evaluating **hunchunk** on its own. Table 13 shows the results of the evaluation for both the original **hunchunk** model and the new hybrid system.

	Precision	Recall	F-score
hunchunk	94.61%	94.88%	94.75%
hunchunk+parser features	95.29%	95.68%	95.48%

Table 13: The role of parser features in base NP chunking

As can be seen from the above figures, the addition of information from a rule-based system leads to a 15% decrease in the number of errors made by the statistical system. We also measured the impact of parser features on a different chunking task which **hunchunk** performs: that of extracting **maximal NPs**, i.e. noun phrases that are not contained by a higher level NP. In the case of maximal noun phrases the parser feature also causes some increase in performance (cf. Table 14).

	Precision	Recall	F-score
hunchunk	89.34%	88.12%	88.72%
hunchunk+parser features	89.46%	88.76%	89.11%

Table 14: The role of parser features in maxNP-chunking

6 Conclusion

This paper described the process of implementing a grammar for Hungarian noun phrases to create an NP-parser and using its output to enhance the performance of a state-of-the-art statistical system. Firstly, we described the technical preliminaries of implementing a context-free grammar and also documented additions and amendments made to both the data and the grammar. Having reached a sufficient parsing quality we proceeded to use the output of the rule-based system to create new features for use with the learning-based **hunchunk** system.

The improved F-scores indicate that hybrid systems in NP-extraction may produce results superior to those of a stand-alone machine learning system. However, it falls beyond the scope of this paper to explore the various possibilities of combining rule-based and statistical approaches to NP-chunking. Also, cross-analysis of errors made by each system – possibly on larger corpora – could help us gain a better understanding of what the strengths and weaknesses of the two approaches are.

References

- [1] Abeillé, A., editor. *Treebanks: Building and using parsed corpora*. Kluwer, Dodrecht, 2003.
- [2] Abney, S. Parsing by chunks. In Berwick, Robert, Abney, Steven, and Tenny, Carol, editors, *Principle-based parsing*, pages 257–278. Kluwer Academic Publishers, 1991.
- [3] Bird, S., Klein, E., and Loper, E. *Natural language processing with Python*. O'Reilly Media, 2009.
- [4] Brill, E. A simple rule-based part of speech tagger. In *Third Conference on Applied Natural Language processing*, pages 152–155, Trento, Italy, 1992.
- [5] Brill, E. Automatic grammar induction and parsing free text: A transformation-based approach, 1993. *Proceedings of the Workshop on Human Language Technology*, pages 237–242, Association for Computational Linguistics, Stroudsburg, PA, USA, 1993.
- [6] Charniak, E. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference (NAACL-2000)*, pages 132–139. Association for Computational Linguistics, 2000.
- [7] Csendes, D., Csirik, J., Gyimóthy, T., and Kocsor, A. The Szeged Treebank. In *Lecture Notes in Computer Science: Text, Speech and Dialogue*, pages 123–131. Springer, 2005.

- [8] Erjavec, T. MULTEXT-east version 3: Multilingual morphosyntactic specifications, lexicons and corpora. In Lino, Maria Teresa, Xavier, Maria Francisca, Ferreira, Fátima, Costa, Rute, and Sila, Raquel, editors, *Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 1535–1538, Paris, 2004. European Language Resources Association (ELRA).
- [9] Gee, J. P. and Grosjean, F. Performance structures: A psycholinguistic and linguistic appraisal. *Cognitive Psychology*, 15:411–458, 1983.
- [10] Hollingshead, K., Fisher, S., and Roark, B. Comparing and combining finite-state and context-free parsers. In *Proceedings of the Conference on Human Language Technologies and Empirical Methods in Natural Language Processing (HLT-EMNLP)*, pages 787–794, 2005.
- [11] Kornai, A. The internal structure of Noun Phrases. In *Approaches to Hungarian*, 1:79–92, 1985.
- [12] Kornai, A. A főnévi csoport egyeztetése. In Kiefer, Ferenc, editor, *Általános Nyelvészeti Tanulmányok*, volume 17, pages 183–211. Akadémiai Kiadó, 1989.
- [13] Kudo, T. and Matsumoto, Y. Use of support vector learning for chunk identification. In *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning - Volume 7*, page 144. Association for Computational Linguistics, 2000.
- [14] Kudo, T. and Matsumoto, Y. Chunking with support vector machines. In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics (NAACL 2001)*, pages 1–8. Association for Computational Linguistics, 2001.
- [15] Lafferty, J., McCallum, A., and Pereira, F. *Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data*. Morgan Kaufmann, 2001.
- [16] Makhoul, J., Kubala, F., Schwartz, R., and Weischedel, R. Performance measures for information extraction. In *Proceedings of the DARPA Broadcast News Workshop*, page 249, Herndon, VA, 1999. Morgan Kaufmann.
- [17] Marcus, M., Santorini, B., and Marcinkiewicz, M. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19:313–330, 1993.
- [18] Ramshaw, L. and Marcus, M. Text chunking using transformation-based learning. In Yarowsky, D. and Church, K., editors, *Proceedings of the Third Workshop on Very Large Corpora*, pages 82–94, Cambridge, MA, 1995. SIG-DAT/ACL.
- [19] Adwait Ratnaparkhi. *Maximum Entropy Models for Natural Language Ambiguity Resolution*. PhD thesis, University of Pennsylvania, 1998.

- [20] Rebrus, P., Kornai, A., and Varga, D. Egy általános célú morfológiai annotáció. [A general purpose morphological annotation system] *Általános Nyelvészeti Tanulmányok*, volume 24, pages 47-80. Akadémiai Kiadó 2012.
- [21] Recski, G. Egy általános célú morfológiai annotáció kiterjesztése [Extending a general-purpose morphological annotation system]. In Váradí, T., editor, *VII. Alkalmazott Nyelvészeti Doktoranduszkonferencia*. pages 168-174. Budapest, MTA Nyelvtudományi Intézet, 2013.
- [22] Recski, G., Rung, A., Zséder, A., and Kornai, A. Np alignment in bilingual corpora. In Calzolari, Nicoletta, Choukri, Khalid, Maegaard, Bente, Mariani, Joseph, Odijk, Jan, Piperidis, Stelios, Rosner, Mike, and Tapias, Daniel, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, 2010. European Language Resources Association (ELRA).
- [23] Recski, G. and Varga, D. A Hungarian NP Chunker. *The Odd Yearbook. ELTE SEAS Undergraduate Papers in Linguistics*, pages 87-93, 2009.
- [24] Recski, G. and Varga, D. Magyar főnévi csoportok azonosítása. *Általános Nyelvészeti Tanulmányok*, volume 24, pages 81-95. Akadémiai Kiadó 2012.
- [25] Recski, G., Varga, D., Zséder, A., and Kornai, A. Főnévi csoportok azonosítása magyar-angol párhuzamos korpuszban [Identifying noun phrases in a parallel corpus of English and Hungarian]. *VI. Magyar Számítógépes Nyelvészeti Konferencia [6th Hungarian Conference on Computational Linguistics]*, 2009.
- [26] Sha, F. and Pereira, F. Shallow parsing with conditional random fields. In *NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 134-141. Association for Computational Linguistics, 2003.
- [27] Sun, X., Morency, L.-P., Okanohara, D., and Tsujii, J. Modeling latent-dynamic in shallow parsing: a latent conditional model with improved inference. In *COLING '08: Proceedings of the 22nd International Conference on Computational Linguistics*, pages 841-848. Association for Computational Linguistics, 2008.
- [28] Tjong Kim Sang, E. F. and Buchholz, S. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning-Volume 7*, pages 127-132. Association for Computational Linguistics, 2000.
- [29] Trón, V., Gyepesi, Gy., Halácsy, P., Kornai, A., Németh, L., and Varga, D. Hunmorph: open source word analysis. In Jansche, Martin, editor, *Proceedings of the ACL 2005 Software Workshop*, pages 77-85. ACL, Ann Arbor, 2005.

A Final grammar of the NP parser

NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NOUN[PRON=POS]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e] ->
 NOUN[PRON=DEM, BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d]
 ART NOUN[PRON=0, BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=0]
 NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NOUN[PRON=GEN]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NOUN[PRON=INDEF]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 ADJ
 NOUN[BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NOUN[BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0] ->
 ADJ
 NOUN[BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0]
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0] ->
 NOUN[BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0]
 NOUN[BAR=2, POSS=?a, PLUR=0, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NUM
 NOUN[BAR=1, POSS=?a, PLUR=0, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
 NOUN[BAR=2, POSS=?a, PLUR=0, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0] ->
 NUM
 NOUN[BAR=1, POSS=?a, PLUR=0, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0]
 NOUN[BAR=2, POSS=?b, PLUR=0, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0] ->
 NOUN[BAR=1, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e]/NOUN[BAR=0, PRON=?f]
 NOUN[BAR=3, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f] ->
 ART[D=?e]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, PRON=?f]
 NOUN[BAR=3, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=1, PRON=?f] ->
 NOUN[BAR=0, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=1, PRON=?f]
 NOUN[BAR=3, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]/NOUN[BAR=0] ->
 ART[D=?e]
 NOUN[BAR=2, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, PRON=?f]/NOUN[BAR=0]
 NOUN[BAR=3, POSS=0, PLUR=?a, ANP=?b, CAS=?c, D=1, PRON=?f] ->
 NOUN[BAR=3, ANP=0, CAS=0]
 NOUN[BAR=2, POSS=1, PLUR=?a, ANP=?b, CAS=?c, PRON=?f]
 NOUN[BAR=4, POSS=0, PLUR=?a, ANP=?b, CAS=?c, D=1, PRON=?f] ->
 NOUN[BAR=3, CAS=[DAT=1]]
 NOUN[BAR=3, POSS=1, PLUR=?a, ANP=?b, CAS=?c, D=1, PRON=?f]
 NOUN[BAR=3, POSS=0, PLUR=?a, ANP=?b, CAS=?c, D=1, PRON=?f] ->

```

ART[BAR=1, D=1, ME=?d, YOU=?e, PLUR=?f]
NOUN[BAR=2, POSS=[ME=?d,YOU=?e,PLUR=?f], PLUR=?a,ANP=?b,CAS=?c,PRON=?f]
NOUN[BAR=3, POSS=0, PLUR=?a, ANP=?b, CAS=?c, D=1, PRON=?f] ->
  ART[BAR=0]
  NOUN[BAR=2, POSS=[], PLUR=?a, ANP=?b, CAS=?c, PRON=?f]
NOUN[POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f, BAR=?g] ->
  PUNCT[TYPE='DQUOTE']
  NOUN[BAR=?g, POSS=?a, PLUR=?b, ANP=?c, CAS=?d, D=?e, PRON=?f]
  PUNCT[TYPE='DQUOTE']
NOUN/NOUN ->

ART[BAR=1, D=1, ME=?a, YOU=?b, PLUR=?c, PRON=?f] ->
  ART[D=1] PRO[ME=?a, YOU=?b, PLUR=?c, PRON=?f]
ART[D=1] -> DET
ADJ -> ADJ ADJ
ADJ -> ADV ADJ
ADJ -> NOUN ADJ[SRC=[STEM=VERB[], DERIV='PERF_PART']]
ADJ -> NOUN ADJ[SRC=[STEM=VERB[], DERIV='IMPERF_PART']]
ADJ -> PUNCT[TYPE='DQUOTE'] ADJ PUNCT[TYPE='DQUOTE']
ADJ -> ADJ PUNCT[TYPE=COMMA] ADJ
ADJ -> ADJ PUNCT[TYPE=COMMA] CONJ ADJ
NUM -> NUM NUM
NUM -> ADV NUM
NUM -> ADJ NUM

```

Received 15th June 2011

Applications of the Inverse Theta Number in Stable Set Problems

Miklós Ujvári*

Abstract

In the paper we introduce a semidefinite upper bound on the square of the stability number of a graph, the inverse theta number, which is proved to be multiplicative with respect to the strong graph product, hence to be an upper bound for the square of the Shannon capacity of the graph. We also describe a heuristic algorithm for the stable set problem based on semidefinite programming, Cholesky factorization, and eigenvector computation.

Keywords: Shannon capacity, stability number, inverse theta number

1 Introduction

An algorithm for the stable set problem is useful in many ways, e.g. it can be used for colouring a graph: find a stable set, remove it from the graph, and iterate the algorithm. (See [2] for further applications and approximation algorithms for the stable set problem.) The strength of the semidefinite programming approach for the stable set and colouring problems is shown by the algorithms of Grötschel–Lovász–Schrijver, Karger–Motwani–Sudan, and Alon–Kahale, see [5] for a summary of these results. In this paper we will describe a heuristic algorithm for the stable set problem based on semidefinite optimization, and the notion of the inverse theta number.

We start the paper with stating the main results. First we fix some notation.

Let $n \in \mathcal{N}$, and let $G = (V(G), E(G))$ be an undirected graph, with vertex set $V(G) = \{1, \dots, n\}$, and with edge set $E(G) \subseteq \{\{i, j\} : i \neq j\}$. Let $A(G)$ be the 0-1 adjacency matrix of the graph G , that is let

$$A(G) := (a_{ij}) \in \{0, 1\}^{n \times n}, \text{ where } a_{ij} := \begin{cases} 0, & \text{if } \{i, j\} \notin E(G), \\ 1, & \text{if } \{i, j\} \in E(G). \end{cases}$$

The complementary graph \overline{G} is the graph with adjacency matrix

$$A(\overline{G}) := J - I - A(G),$$

*H-2600 Vác, Szent János utca 1., Hungary. E-mail: ujvarim@cs.elte.hu

where I is the identity matrix, and J denotes the matrix with all elements equal to one. The disjoint union of the graphs G_1 and G_2 is the graph $G_1 + G_2$ with adjacency matrix

$$A(G_1 + G_2) := \begin{pmatrix} A(G_1) & 0 \\ 0 & A(G_2) \end{pmatrix}.$$

We will use the notation K_n for the clique graph, and K_{s_1, \dots, s_k} for the complete multipartite graph $K_{s_1} + \dots + K_{s_k}$. Also, we will denote by C_n the n -cycle, the polygon graph with n vertices.

Let $(\delta_1, \dots, \delta_n)$ be the sum of the row vectors of the adjacency matrix $A(G)$. The elements of this vector are the degrees of the vertices of the graph G . Let $\delta_G, \Delta_G, \mu_G$ be the minimum, maximum, resp. the arithmetic mean of the degrees in the graph.

By Rayleigh's theorem (see [9]) for a symmetric matrix $M = M^T \in \mathcal{R}^{n \times n}$ the minimum and maximum eigenvalue, λ_M , resp. Λ_M can be expressed as

$$\lambda_M = \min_{\|u\|=1} u^T M u, \quad \Lambda_M = \max_{\|u\|=1} u^T M u.$$

Attainment occurs if and only if $u \in \mathcal{R}^n$ is a unit eigenvector corresponding to λ_M and Λ_M , respectively. The minimum and maximum eigenvalue of the adjacency matrix $A(G)$ will be denoted by λ_G , resp. Λ_G .

The set of the n by n real symmetric positive semidefinite matrices will be denoted by \mathcal{S}_+^n , that is

$$\mathcal{S}_+^n := \{M \in \mathcal{R}^{n \times n} : M = M^T, u^T M u \geq 0 \ (u \in \mathcal{R}^n)\}.$$

For example, the Laplacian matrix of the graph G ,

$$L(G) := D_{\delta_1, \dots, \delta_n} - A(G) \in \mathcal{S}_+^n.$$

(Here $D_{\delta_1, \dots, \delta_n}$ denotes the diagonal matrix with diagonal elements $\delta_1, \dots, \delta_n$.)

It is well-known (see [9]), that the following statements are equivalent for a symmetric matrix $M = (m_{ij}) \in \mathcal{R}^{n \times n}$: a) $M \in \mathcal{S}_+^n$; b) $\lambda_M \geq 0$; c) M is Gram matrix, that is $m_{ij} = v_i^T v_j$ ($i, j = 1, \dots, n$) for some vectors v_1, \dots, v_n . Furthermore, by Lemma 2.1 in [13], the set \mathcal{S}_+^n can be described as

$$\mathcal{S}_+^n = \left\{ \left(\frac{a_i^T a_j}{(a_i a_j^T)_{11}} - 1 \right)_{i,j=1}^n \mid \begin{array}{l} d \in \mathcal{N}, a_i \in \mathcal{R}^d \ (1 \leq i \leq n) \\ a_i^T a_i = 1 \ (1 \leq i \leq n) \end{array} \right\}. \quad (1)$$

The stability number, $\alpha(G)$, is the maximum cardinality of the (so-called stable) sets $S \subseteq V(G)$ such that $\{i, j\} \subseteq S$ implies $\{i, j\} \notin E(G)$. The chromatic number, $\chi(G)$, is the minimum number of stable sets covering the vertex set $V(G)$.

Let us define an *orthonormal representation* of the graph G (shortly, o.r. of G) as a system of vectors $a_1, \dots, a_n \in \mathcal{R}^d$ for some $d \in \mathcal{N}$, satisfying

$$a_i^T a_i = 1 \ (i = 1, \dots, n), \quad a_i^T a_j = 0 \ (\{i, j\} \in E(\overline{G})).$$

In the seminal paper [6] L. Lovász proved the following result, now popularly called *sandwich theorem*, see [4]:

$$\alpha(G) \leq \vartheta(G) \leq \chi(\overline{G}),$$

where $\vartheta(G)$ is the *Lovász number* of the graph G , defined as

$$\vartheta(G) := \inf \left\{ \max_{1 \leq i \leq n} \frac{1}{(a_i a_i^T)_{11}} : a_1, \dots, a_n \text{ o.r. of } G \right\}.$$

The Lovász number has several equivalent descriptions, see [6]. For example, by (1) and standard semidefinite duality theory (see e.g. [12]), it is the common optimal value of the Slater-regular primal-dual semidefinite programs

$$(TP) \quad \min \lambda, \begin{cases} x_{ii} = \lambda - 1 \ (i \in V(G)), \\ x_{ij} = -1 \ (\{i, j\} \in E(\overline{G})), \\ X = (x_{ij}) \in \mathcal{S}_+^n, \lambda \in \mathcal{R} \end{cases}$$

and

$$(TD) \quad \max \operatorname{tr}(JY), \begin{cases} \operatorname{tr}(Y) = 1, \\ y_{ij} = 0 \ (\{i, j\} \in E(G)), \\ Y = (y_{ij}) \in \mathcal{S}_+^n. \end{cases}$$

(Here tr stands for trace.) Reformulating the program (TD) , Lovász derived the following dual description of the theta number (Theorem 5 in [6]):

$$\vartheta(G) = \max \left\{ \sum_{i=1}^n (b_i b_i^T)_{11} : b_1, \dots, b_n \text{ o.r. of } \overline{G} \right\}. \quad (2)$$

An important application of the theory of the theta number is described in Theorem 1 of [6], where it is proved that

$$\Theta(G) \leq \vartheta(G), \quad (3)$$

with $\Theta(G)$ denoting the *Shannon capacity* of the graph, that is

$$\Theta(G) := \sup_{k \in \mathcal{N}} \sqrt[k]{\alpha(G^k)}.$$

(Here $G \cdot H$ denotes the strong graph product of the graphs G, H , the graph with vertex set

$$V(G \cdot H) := \{(i, j) : i \in V(G), j \in V(H)\}$$

and edge set

$$E(G \cdot H) := \left\{ \{(i_1, j_1), (i_2, j_2)\} \mid \begin{array}{l} i_1 = i_2 \text{ or } \{i_1, i_2\} \in E(G) \\ j_1 = j_2 \text{ or } \{j_1, j_2\} \in E(H) \end{array} \right\}.$$

Also, G^k denotes the strong graph product of k copies of the graph G .)

The proof of (3) relies on the fact that the theta function $\vartheta(\cdot)$ is submultiplicative, that is

$$\vartheta(G \cdot H) \leq \vartheta(G) \cdot \vartheta(H)$$

holds for any graphs G, H . Another two submultiplicative bounds are described in [6], see Theorems 10 and 11; they turn out to be weaker than the theta number.

In Section 2 we will define the inverse theta number as

$$\iota(G) := \inf \left\{ \sum_{i=1}^n \frac{1}{(a_i a_i^T)_{11}} : a_1, \dots, a_n \text{ o.r. of } G \right\},$$

and derive the inequality

$$\alpha(G) \leq \sqrt{\iota(G)},$$

an analogue of Lovász's sandwich theorem. In Section 3 we will prove also (as a consequence of multiplicativity properties) the stronger relation

$$\Theta(G) \leq \sqrt{\iota(G)}. \quad (4)$$

It is known (see Proposition 2.2) that e.g. for the cycle graphs C_n , $\sqrt{\iota(C_n)} > \vartheta(C_n)$ holds. Hence, the inverse theta number does not help in determining the Shannon capacity of the odd cycles C_7, C_9, \dots , which is still an open problem, though, using the theta number, Lovász determined the Shannon capacity of the 5-cycle and other graphs in [6]. However, we will see in Section 4, that orthonormal representations of the complementary graph \overline{G} of high value in the dual description (5) of the inverse theta number, can be of use in a heuristic algorithm calculating large stable sets in any graph G .

2 The inverse theta function

The inverse theta number is defined via optimizing over the inverse of the theta body.

The reformulation of $\vartheta(G)$ described in (2) can be written concisely, as

$$\vartheta(G) = \max \left\{ \sum_{i=1}^n y_i : y = (y_i) \in TH(G) \right\},$$

where $TH(G)$ denotes the *theta body*, that is the set of vectors $y = (y_i) \in \mathcal{R}^n$ such that $y_i = (b_i b_i^T)_{11}$ ($i = 1, \dots, n$) for some orthonormal representation (b_i) of the complementary graph \overline{G} .

Convexity and compactness of the theta body follows from the fact (see Corollary 29 of [4]), that $TH(G)$ can be described equivalently as the set of vectors $y = (y_i) \in \mathcal{R}^n$ for which there exists a matrix $W = (w_{ij}) \in \mathcal{R}^{n \times n}$ satisfying both

$$\begin{pmatrix} 1 & y^T \\ y & W \end{pmatrix} \in \mathcal{S}_+^{n+1},$$

and

$$y_i = w_{ii} \ (i = 1, \dots, n), \ w_{ij} = 0 \ (\{i, j\} \in E(G)).$$

Analogously, let us denote by $TH^-(G)$ the *inverse theta body*, that is the set of vectors $x = (x_i) \in \mathcal{R}^n$ such that $x_i = 1/(a_i a_i^T)_{11}$ ($i = 1, \dots, n$) for some orthonormal representation (a_i) of the graph G .

From (1) it follows immediately, that $TH^-(G)$ can be described equivalently as the set of vectors $x = (x_i) \in \mathcal{R}^n$ such that there exists a matrix $Z = (z_{ij}) \in \mathcal{R}^{n \times n}$ satisfying

$$z_{ii} = x_i - 1 \ (i = 1, \dots, n), \ z_{ij} = -1 \ (\{i, j\} \in E(\overline{G})), \ Z \in \mathcal{S}_+^n.$$

This fact implies the convexity of the inverse theta body, and also its monotonicity: if $\hat{x} \geq x \in TH^-(G)$ then $\hat{x} \in TH^-(G)$, too.

Let us define the *inverse theta number* of a graph G as

$$\iota(G) := \inf \left\{ \sum_{i=1}^n x_i : x = (x_i) \in TH^-(G) \right\}.$$

From the above considerations, and standard semidefinite duality theory (see e.g. [12]) we obtain the following statement, which implies also that the inverse theta number is efficiently computable using interior-point algorithms (see e.g. [7], [1], [10]).

Theorem 2.1. *The inverse theta number $\iota(G)$ equals the common optimal value of the Slater-regular primal-dual semidefinite programs*

$$\begin{aligned} (TP^-) \quad & \inf \operatorname{tr}(Z) + n, \ z_{ij} = -1 \ (\{i, j\} \in E(\overline{G})), \ Z = (z_{ij}) \in \mathcal{S}_+^n, \\ (TD^-) \quad & \sup \operatorname{tr}(JM), \ \begin{cases} m_{ii} = 1 \ (i = 1, \dots, n), \\ m_{ij} = 0 \ (\{i, j\} \in E(G)), \\ M = (m_{ij}) \in \mathcal{S}_+^n. \end{cases} \end{aligned}$$

The optimal values of the programs (TP^-) and (TD^-) are attained. □

Moreover, rewriting the feasible solution M of the program (TD^-) as the Gram matrix $M = (b_i^T b_j)$ for some vectors $b_1, \dots, b_n \in \mathcal{R}^d$, we obtain the following analogue of (2):

$$\iota(G) = \max \left\{ \sum_{i,j=1}^n b_i^T b_j : b_1, \dots, b_n \text{ o.r. of } \overline{G} \right\}. \tag{5}$$

Similarly to $\vartheta(G)$, the number $\iota(G)$ constitutes an upper bound for the stability number $\alpha(G)$.

Theorem 2.2. *For any graph G , $\alpha(G) \leq \sqrt{\iota(G)}$ holds.*

Proof. We adapt the proof of Lemma 3 in [6].

Let $S \subseteq V(G)$ be a stable set, with cardinality $\alpha(G)$. Then for any (a_i) orthonormal representation of G , the vectors a_i ($i \in S$) are pairwise orthogonal unit vectors. Therefore

$$\sum_{i \in S} (a_i a_i^T)_{11} \leq 1,$$

which formula, by the arithmetic-harmonic mean inequality, implies that

$$\sum_{i=1}^n \frac{1}{(a_i a_i^T)_{11}} \geq \sum_{i \in S} \frac{1}{(a_i a_i^T)_{11}} \geq (\alpha(G))^2$$

holds. Taking infimum in (a_i) , we have the statement. \square

The next two propositions give in particular the exact value of $\iota(G)$ for complete multipartite graphs and for graphs with vertex-transitive automorphism group.

Proposition 2.1. *For any graph G , the inequalities*

$$n \left(1 + \frac{\mu_{\overline{G}}}{-\lambda_{\overline{G}}} \right) \leq \iota(G) \leq n(\mu_{\overline{G}} + 1)$$

hold, with equality if G is a complete multipartite graph.

Proof. The inequalities are proved by the feasible solutions

$$Z := L(\overline{G}), \quad M := I + \frac{1}{-\lambda_{\overline{G}}} A(\overline{G}),$$

which matrices have the values

$$n(\mu_{\overline{G}} + 1), \quad n \left(1 + \frac{\mu_{\overline{G}}}{-\lambda_{\overline{G}}} \right)$$

in (TP^-) and (TD^-) , respectively.

The last assertion follows from the fact that for complete multipartite graphs $\lambda_{\overline{G}} = -1$. \square

The following proposition implies that for graphs with vertex-transitive automorphism group $\sqrt{\iota(G)} > \vartheta(G)$.

Proposition 2.2. *For any graph G , the inequalities*

$$\frac{n^2}{\vartheta(G)} \leq \iota(G) \leq n\vartheta(G)$$

hold, with equality if the graph G has vertex-transitive automorphism group.

Proof. First, let (a_i) and (b_i) be orthonormal representations of G and \overline{G} , respectively. Then, by Lemma 4 in [6],

$$\sum_{i=1}^n (a_i a_i^T)_{11} (b_i b_i^T)_{11} \leq 1$$

holds, which formula implies, by the arithmetic-harmonic mean inequality, that

$$\sum_{i=1}^n \frac{1}{(a_i a_i^T)_{11} (b_i b_i^T)_{11}} \geq n^2.$$

Consequently,

$$\max_{1 \leq i \leq n} \frac{1}{(b_i b_i^T)_{11}} \cdot \sum_{i=1}^n \frac{1}{(a_i a_i^T)_{11}} \geq n^2,$$

and taking infimum in (a_i) and (b_i) we have the inequality $\iota(G) \geq n^2/\vartheta(\overline{G})$.

On the other hand, if M is feasible in (TD^-) then $Y = M/n$ is feasible in (TD) , which proves the inequality $\iota(G) \leq n\vartheta(G)$, too.

The last assertion follows from the fact that for graphs with vertex-transitive automorphism group, the equality $\vartheta(G)\vartheta(\overline{G}) = n$ holds (see Theorem 8 in [6]). \square

We conclude this section with an open problem. Closedness of the convex set $TH^-(G)$ follows easily from the fact that $TH(\overline{G})$ is a compact set. Hence, the inverse theta body can be described as

$$TH^-(G) = \bigcap_{w \geq 0} \{x \in \mathcal{R}^n : w^T x \geq \iota(G, w)\},$$

where $\iota(G, w)$ denotes the weighted version of $\iota(G)$, that is

$$\iota(G, w) := \inf\{w^T x : x \in TH^-(G)\} \quad (w \in \mathcal{R}^n).$$

For special vectors $w \in \mathcal{R}^n$, we have seen in the proof of Proposition 2.2 that

$$TH^-(G) \subseteq \bigcap_{(b_i)} \left\{ x = (x_i) \in \mathcal{R}^n : \sum_{i=1}^n \frac{x_i}{(b_i b_i^T)_{11}} \geq n^2, x \geq 0 \right\},$$

where the (b_i) s are the orthonormal representations of the complementary graph \overline{G} . Does equality hold here? (For the theta body a similar linear description is known (see [4]):

$$TH(G) = \bigcap_{(a_i)} \left\{ y = (y_i) \in \mathcal{R}^n : \sum_{i=1}^n (a_i a_i^T)_{11} y_i \leq 1, y \geq 0 \right\},$$

where the (a_i) s are the orthonormal representations of the graph G .)

3 Shannon capacity

In this section we will prove that the inverse theta function has the same multiplicativity properties as the theta function, consequently its square root is an upper bound for the Shannon capacity of the graph.

First, we will verify the submultiplicativity of the inverse theta function, an analogue of Lemma 2 in [6].

Lemma 3.1. *For any graphs G, H , $\iota(G \cdot H) \leq \iota(G) \cdot \iota(H)$.*

Proof. Let (a_i^G) and (a_j^H) be orthonormal representations of the graphs G and H , respectively. Then, by Lemma 1 in [6], $(a_i^G \otimes a_j^H)$ is an orthonormal representation of the graph $G \cdot H$. (Here $x \otimes y$ denotes Kronecker product of the vectors $x = (x_i)$, y , that is the block vector $x \otimes y := (x_i \cdot y)$, see [8].) Thus,

$$\begin{aligned} \iota(G \cdot H) &\leq \sum_{i,j} 1 / ((a_i^G \otimes a_j^H)(a_i^G \otimes a_j^H)^T)_{11} \\ &= \sum_i 1 / (a_i^G a_i^{GT})_{11} \cdot \sum_j 1 / (a_j^H a_j^{HT})_{11}, \end{aligned}$$

and, taking infimum in (a_i^G) and (a_j^H) , we have the statement. \square

Now, we will prove the skew-supermultiplicativity of the inverse theta function.

Lemma 3.2. *For any graphs G, H , $\iota(\overline{G \cdot H}) \geq \iota(G) \cdot \iota(H)$.*

Proof. Let (b_i^G) and (b_j^H) be orthonormal representations of the complementary graphs \overline{G} and \overline{H} , respectively. Then, by Lemma 1 in [6], $(b_i^G \otimes b_j^H)$ is an orthonormal representation of the graph $\overline{G \cdot H}$. Thus, by (5),

$$\begin{aligned} \iota(\overline{G \cdot H}) &\geq \sum_{i_1, i_2, j_1, j_2} (b_{i_1}^G \otimes b_{j_1}^H)^T (b_{i_2}^G \otimes b_{j_2}^H) \\ &= \sum_{i_1, i_2} b_{i_1}^{GT} b_{i_2}^G \cdot \sum_{j_1, j_2} b_{j_1}^{HT} b_{j_2}^H, \end{aligned}$$

and, taking supremum in (b_i^G) and (b_j^H) , the statement is proved. \square

Summarizing, we obtain the following analogue of Theorem 7 in [6].

Theorem 3.1. *The inequalities in Lemmas 3.1 and 3.2 hold with equalities: for any graphs G, H ,*

$$a) \iota(G \cdot H) = \iota(G) \cdot \iota(H);$$

$$b) \iota(\overline{G \cdot H}) = \iota(G) \cdot \iota(H).$$

Proof. It is enough to notice that the graph $G \cdot H$ is a subgraph of $\overline{G \cdot H}$, so

$$\iota(G \cdot H) \geq \iota(\overline{G \cdot H}).$$

Applying Lemmas 3.1 and 3.2, the proof is completed. \square

We remark that part of Theorem 3.1 holds also with $+$ signs instead of \cdot signs:

$$\iota(G + H) \geq \iota(G) + \iota(H) = \iota(\overline{G + H}),$$

for any graphs G, H . The proof of this statement is immediate from Theorem 2.1, therefore it is omitted. (For analogous results with the theta function, see [4].)

A submultiplicative upper bound for the stability number of a graph is also an upper bound for the Shannon capacity of the graph, see Theorem 1 in [6]. Consequently,

Theorem 3.2. *For any graph G , $\Theta(G) \leq \sqrt{\iota(G)}$ holds.*

Proof. By Theorem 2.2, for any graph H , $\alpha(H) \leq \sqrt{\iota(H)}$. Hence, from Lemma 3.1,

$$\alpha(G^k) \leq \sqrt{\iota(G^k)} \leq \left(\sqrt{\iota(G)}\right)^k$$

follows for $k \in \mathcal{N}$; the proof is finished. \square

Summarizing Theorem 1 in [6] and Theorem 3.2 we obtain

$$\Theta(G) \leq \min \left\{ \vartheta(G), \sqrt{\iota(G)} \right\}. \quad (6)$$

Can $\sqrt{\iota(G)}$ be less than $\vartheta(G)$ for some graph G ? Juhász's theorem (see [3]) states that $\vartheta(G)$ is typically "around" $n^{1/2}$ in the following sense:

Theorem 3.3. (Juhász) *Let G be a random graph with edge probability $p = 1/2$. Then, with probability $1 - o(1)$ for $n \rightarrow \infty$,*

$$\frac{1}{2}\sqrt{n} + O(n^{1/3} \log n) \leq \vartheta(G) \leq 2\sqrt{n} + O(n^{1/3} \log n).$$

Hence, the value $\sqrt{\iota(G)}$ (which is between $n / \sqrt{\vartheta(G)}$ and $\sqrt{n\vartheta(G)}$ by Proposition 2.2) is typically "around" $n^{3/4}$.

Theorem 3.4. *Let G be a random graph with edge probability $p = 1/2$. Then, there exist positive constants $c_1, c_2 > 0$ such that with probability $1 - o(1)$ for $n \rightarrow \infty$,*

$$c_1 \cdot n^{3/4} \leq \sqrt{\iota(G)} \leq c_2 \cdot n^{3/4}.$$

(Any $c_1, c_2 > 0$ such that $c_1^2 < 1/2$ and $c_2^2 > 2$ meet the requirements.) \square

We mention two corollaries: a positive and a negative result with non-constructive proofs.

Corollary 3.1. *There exist graphs G such that $\sqrt{\iota(G)} < \chi(G)$.*

Proof. The proof is indirect: Let us suppose that the inequality

$$\chi(H) \leq \sqrt{\iota(\overline{H})}$$

holds for any graph H .

Then, by Theorem 3.4,

$$\alpha(H) \geq \frac{n}{\chi(H)} \geq \frac{n}{\sqrt{\iota(\overline{H})}} \geq c \cdot n^{1/4} \quad (7)$$

would hold, with probability $1 - o(1)$ as $n \rightarrow \infty$, for some appropriate constant $c > 0$. On the other hand, it can easily be seen that the probability of $\alpha(H) \geq \ell$,

$$\begin{aligned} P(\alpha(H) \geq \ell) &\leq \binom{n}{\ell} \cdot \left(1 - \frac{1}{2}\right)^{\ell(\ell-1)/2} \leq \\ &\leq \left(n \cdot 2^{-(\ell-1)/2}\right)^{\ell} \rightarrow 0 \quad (n \rightarrow \infty), \end{aligned}$$

where $\ell := c \cdot n^{1/4}$. We reached contradiction with (7).

Hence, there exist graphs satisfying

$$\sqrt{\iota(\overline{H})} < \chi(H),$$

from which, with $G = \overline{H}$, the statement follows. \square

From Theorems 3.3 and 3.4 immediately follows

Corollary 3.2. *Under the assumptions of Theorems 3.3 and 3.4, with probability $1 - o(1)$ for $n \rightarrow \infty$,*

$$\vartheta(G) \leq \sqrt{\iota(G)}.$$

\square

Thus, the graphs G , with $\sqrt{\iota(G)} < \vartheta(G)$, if they exist at all, are rare. However, we will see in the following section, that the fact that $\iota(G)$ with high probability is large, can be an advance, too.

We conclude this section with an open problem: With minor modification of the proof of Theorem 2.2 it can be proved that

$$\alpha(G)^2 \leq \iota(G) - n + \alpha(G).$$

From this inequality we obtain the bound

$$\alpha(G) \leq \frac{1}{2} \left(1 + \sqrt{4(\iota(G) - n) + 1}\right), \quad (8)$$

which is tighter than $\alpha(G) \leq \sqrt{\iota(G)}$. It is an open problem, whether the bound in (8) is submultiplicative (and, thus, is an upper bound for the Shannon capacity $\Theta(G)$), or not.

4 Heuristic algorithm

In this section we will describe a heuristic algorithm for the stable set problem.

The key observation for the algorithm is the following simple

Lemma 4.1. *Let the vectors $b_1, \dots, b_n \in \mathcal{R}^d$ form an orthonormal representation of the complementary graph \overline{G} , and let $u \in \mathcal{R}^d$, $u^T u = 1$. Then,*

$$S := \left\{ i \in \{1, \dots, n\} : (u^T b_i)^2 > \frac{1}{2} \right\} \quad (9)$$

is a stable set in the graph G .

Proof. Let us suppose indirectly that for some $i, j \in S$, $\{i, j\} \in E(G)$. Then, as (b_1, \dots, b_n) is an orthonormal representation of \overline{G} , so $b_i^T b_j = 0$, and $\|b_i + b_j\| = \sqrt{2}$. By $i, j \in S$, we have $(u^T b_i)^2 > 1/2 < (u^T b_j)^2$. Let us consider for example the case when $u^T b_i > \sqrt{2}/2 < u^T b_j$. Then,

$$\sqrt{2} < u^T (b_i + b_j) \leq \|u\| \cdot \|b_i + b_j\| = \sqrt{2},$$

which is a contradiction. The cases, when $u^T b_i < -\sqrt{2}/2$ or $u^T b_j < -\sqrt{2}/2$ can be dealt with similarly. This completes the proof. \square

Taking into account Lemma 4.1 we can search for large stable sets as follows: We compute an orthonormal representation (b_i) of the complementary graph \overline{G} and a unit vector u so that $\sum_i (u^T b_i)^2$ is maximal, that is, see (2), it equals $\vartheta(G)$. (The solution of this problem is well-known, see Theorem 12 in [5].) The output stable set S will be the one in (9). The algorithm derived this way is a special case of the Alon-Kahale algorithm, see Theorem 29 in [5].

To calculate with the inverse theta function $\iota(G)$ instead of the theta number $\vartheta(G)$, we take a different approach to the problem. It follows from Rayleigh's theorem and (2) that finding an orthonormal representation (b_i) of the complementary graph \overline{G} and unit vector u with value $\sum (u^T b_i)^2 = \vartheta(G)$ means solving the programs

$$(P_d) \quad \sup \Lambda_{BB^T}, \quad \begin{cases} (B^T B)_{ii} = 1 \quad (i = 1, \dots, n) \\ (B^T B)_{ij} = 0 \quad (\{i, j\} \in E(G)), \end{cases}$$

where $B = (b_1, \dots, b_n) \in \mathcal{R}^{d \times n}$. In other words, using the obvious equality $\Lambda_{BB^T} = \Lambda_{B^T B}$ and the variable transformation $M = B^T B$, we have to solve the program

$$(P) \quad \sup \Lambda_M, \quad \begin{cases} m_{ii} = 1 \quad (i = 1, \dots, n) \\ m_{ij} = 0 \quad (\{i, j\} \in E(G)) \\ M = (m_{ij}) \in \mathcal{S}_+^n. \end{cases}$$

This reformulation with a different proof is due to L. Lovász, who proved also the equivalence of the programs (P) and (TD) , see [11], Theorems 11.8 and 11.3.

Algorithm 1 Heuristic algorithm for the stable set problem.

- 1: Solve to optimality (or with $\varepsilon > 0$ additive error) the program (TD^-) . Denote the solution by M^* . (The ε -optimal solution M^* can be determined in polynomial time using interior-point methods for semidefinite optimization, see e.g. [7], [1], [10].)
 - 2: Determine a matrix $B = (b_1, \dots, b_n) \in \mathcal{R}^{d \times n}$ such that $M^* = B^T B$. (An appropriate matrix B can be determined in polynomial time using algorithms from [9], e.g. Cholesky factorization.)
 - 3: Compute a vector $u \in \mathcal{R}^d$, $u^T u = 1$ such that $\Lambda_{BB^T} = u^T B B^T u$ holds. In other words compute a unit eigenvector of the matrix $B B^T$ corresponding to its maximum eigenvalue Λ_{BB^T} . (This can be accomplished in polynomial time using algorithms from [9].)
 - 4: Return the stable set S in (9).
-

To obtain an algorithm based on the notion of the inverse theta number, instead of (P) we solve the program (TD^-) for M , and from this matrix we compute B , u and the stable set S . The algorithm derived this way is as follows:

We have some evidence that our algorithm finds large stable sets. Note that the following theorem implies, by Juhász's theorem, that $\sum_i (u^T b_i)^2$ is typically "around" \sqrt{n} for the modified algorithm, similarly as in the case of its original version, the Alon-Kahale algorithm.

Theorem 4.1. *Algorithm 1 computes an orthonormal representation (b_1, \dots, b_n) of the complementary graph \overline{G} , and a unit vector $u \in \mathcal{R}^d$ such that the inequalities*

$$\vartheta(G) \geq \sum_{i=1}^n (u^T b_i)^2 \geq \frac{\iota(G)}{n} \geq \frac{n}{\vartheta(\overline{G})}$$

hold.

Proof. The first inequality is the immediate consequence of Theorem 5 in [6]. Let us prove the second inequality. Obviously,

$$\sum_{i=1}^n (u^T b_i)^2 = \Lambda_{BB^T} = \Lambda_{B^T B} = \Lambda_{M^*}.$$

On the other hand, by Rayleigh's theorem,

$$\Lambda_{M^*} \geq \frac{\mathbf{1}^T}{\sqrt{n}} M^* \frac{\mathbf{1}}{\sqrt{n}} = \frac{\text{tr}(J M^*)}{n} = \frac{\iota(G)}{n},$$

where $\mathbf{1}$ denotes the n -vector with all elements equal to one. This way we have verified the inequality $\sum_i (u^T b_i)^2 \geq \iota(G)/n$. Finally, the last inequality follows from Proposition 2.2. \square

Note that the following corollary of Theorem 4.1 implies the relation

$$\alpha(G) \geq \frac{2\iota(G)}{n} - n. \quad (10)$$

(Similarly,

$$\alpha(G) \geq 2\vartheta(G) - n,$$

as the Alon-Kahale algorithm shows.)

Corollary 4.1. *Algorithm 1 realizes the bound in (10): finds a stable set S with cardinality $|S| \geq (2\iota(G)/n) - n$.*

Proof. The statement is an easy consequence of the inequality

$$\sum_{i \in S} (u^T b_i)^2 + \sum_{i \notin S} (u^T b_i)^2 \geq \frac{\iota(G)}{n},$$

as for $i \notin S$ we have $(u^T b_i)^2 \leq 1/2$ by the definition of the stable set S in (9). \square

Corollary 4.1 implies that $|S| > 0$ if $\iota(G) > n^2/2$. Thus, the output stable set S is nonempty for example when $\alpha(G) > n/\sqrt{2}$.

We conclude this section with a simple example. Let us consider the graph $G = K_{s_1, \dots, s_k}$. Then, the output matrix M^* (the optimal solution of the program (TD^-)) is the block-diagonal matrix made up of the matrices $J \in \mathcal{R}^{s_1 \times s_1}, \dots, J \in \mathcal{R}^{s_k \times s_k}$ as diagonal blocks, zero otherwise. The matrix $B \in \mathcal{R}^{k \times n}$ such that $M^* = B^T B$ is made up of the column vectors of the identity matrix $I \in \mathcal{R}^{k \times k}$ with multiplicity s_1, \dots, s_k , respectively. Then, $BB^T \in \mathcal{R}^{k \times k}$ is the diagonal matrix with diagonal elements s_1, \dots, s_k . Let us suppose that $s_1 \geq s_2, \dots, s_k$. Then, the vector $u \in \mathcal{R}^k$ equals the first column vector of the identity matrix $I \in \mathcal{R}^{k \times k}$; and $S = \{1, \dots, s_1\}$ is the output stable set.

We can see that our heuristic algorithm in the case of the graph $G = K_{s_1, \dots, s_k}$ finds a maximum stable set (and, iterating the algorithm, we obtain a minimum colouring). Generally, estimating from below the factor of the algorithm, the infimum ratio of the cardinality of the output stable set and the stability number for a graph with n vertices, is an unsolved problem.

5 Conclusion

In this paper we studied the multiplicativity properties of the inverse theta function, and as a consequence we proved that the square root of this function is an upper bound for the Shannon capacity of the graph. Though the square root of the inverse theta number, as compared to Lovász's theta number, is typically a weak upper bound, this fact could be exploited in a heuristic algorithm for the stable set problem.

References

- [1] de Klerk, E. *Interior Point Methods for Semidefinite Programming*. PhD thesis, Technische Universiteit Delft, 1997.
- [2] Halldórsson, M.M. Approximations of independent sets in graphs. In: Jansen, K. and Rolim, J., editors., APPROX '98, *Lecture Notes in Computer Science*, 1444:1–13, 1998.
- [3] Juhász, F. The asymptotic behaviour of Lovász' ϑ function for random graphs. *Combinatorica* 2:153–155, 1982.
- [4] Knuth, D. The sandwich theorem. *Electronic Journal of Combinatorics*, 1:1–48, 1994.
- [5] Laurent, M. and Rendl, F. Semidefinite programming and integer programming. In: Aardal, K. et al., editors., *Handbook on Discrete Optimization*, Elsevier B.V., Amsterdam, pages 393–514, 2005.
- [6] Lovász, L. On the Shannon capacity of a graph. *IEEE Transactions on Information Theory*, IT-25:1–7, 1979.
- [7] Nesterov, Y. and Nemirovskii, A. *Interior-Point Polynomial Methods in Convex Programming*. Studies in Applied Mathematics 13, SIAM, Philadelphia, 1994.
- [8] Praszolov, V.V. *Lineáris Algebra*. Typotex Kiadó, Budapest, 2005.
- [9] Strang, G. *Linear Algebra and its Applications*. Academic Press, New York, 1980.
- [10] Sturm, J.F. *Primal-Dual Interior Point Approach to Semidefinite Programming*. PhD thesis, Tinbergen Institute Research Series 156, Thesis Publishers, Amsterdam, 1997.
- [11] Ujvári, M. *A Szemidefinit Programozás Alkalmazásai a Kombinatorikus Optimalizálásban*. Eötvös Kiadó, Budapest, 2001.
- [12] Ujvári, M. A note on the graph-bisection problem. *Pure Mathematics and Applications* 12(1):119–130, 2002.
- [13] Ujvári, M. New descriptions of the Lovász number, and the weak sandwich theorem. *Acta Cybernetica* 20(4):499–513, 2012.

Received 9th April 2013

Time-dependent Network Algorithm for Ranking in Sports

András London, József Németh, and Tamás Németh*

Abstract

In this paper a novel ranking method which may be useful in sports like tennis, table tennis or American football, etc. is introduced and analyzed. In order to rank the players or teams, a time-dependent PageRank based method is applied on the directed and weighted graph representing game results in a sport competition. The method was examined on the results of the table tennis competition of enthusiastic sport-loving researchers of the Institute of Informatics at the University of Szeged. The results of our method were compared by several popular ranking techniques. We observed that our approach works well in general and it has a good predictive power.

Keywords: Colley method, Least squares method, Keener method, Markov chain, PageRank, ranking algorithms, self-organization

1 Introduction

In the last decade, rating and ranking methods have been studied and applied in a wide range of different areas. Due to the extraordinary success of Google's PageRank (PR) algorithm [7] -originally developed for ranking webpages based on their importance- graph based algorithms have gained more ground in the topic of ranking problems. Some good surveys on the PageRank method can be found in [4, 20, 29]. Recently, the dynamic extensions of the PageRank method have also been discussed, containing the dynamic aspects of the 'damping' factor [28] and the viewpoint of the evolving network [3] and the time dependency [2]. More recently, a novel dynamic ranking model has also been proposed for ranking in sports [23].

Ranking athletes in individual sports, or sport teams is important for those who are interested in the various professional or amateur leagues as a financial investor, a manager or a fan and it also has a crucial role in sports betting from the point of view of both the better and the betting agency. In many sports, only the win/loss ratio is considered (*e.g.* see the most popular sports in the U.S.) for ranking, *i.e.* higher value indicates higher position in the ranking. In the case of equal win/loss rates, the result(s) of the head-to-head matches

*University of Szeged, Department of Computer Science
E-mail: {london, nemjzozs, tnemeth}@inf.u-szeged.hu

between the players/teams in question and other simple statistics are considered to determine the ranking positions. In many sports, instead of the round-robin system, the type of the most relevant competitions is a single-elimination tournament (also called knock-out or cup) maybe with a preceding group stage. Thus the players play just few matches against only a small subset of the competitors. The official ranking of the players is usually determined by a sport specific rating system (*e.g.* see tennis, table tennis, combat sports, etc.). In fact, in a tournament, in a regular season or in a given period each player/team plays with only a subset of the others and a player/team who plays against weaker opponents have a considerable advantage compared with those, who play against stronger ones.

Many approach traces back the ranking problem to the solution of a system of linear equations, where the entries of the coefficient matrix refer in some way to the results of the games have been played. Due to the study of this pairwise comparison scheme (for early studies see *e.g.* [6, 12, 19]), several matrix-based ranking algorithm have been appeared related to the ranking in sports (see *e.g.* [10] for chess teams, [11, 26] for tennis players, [5, 8, 14, 21] for American football teams). For a good mathematical guide to sports, see *e.g.* [16], while a useful comprehensive work can be found in [13] and [21].

In this paper, we continue this direction of studies and present a simple, time-dependent PageRank based method, the *time-dependent PageRank* (tdPR), and apply it to the table tennis competition of the Institute of Informatics at the University of Szeged. In that competition, there is no any regular organization rule: players play against any participant whenever they want. Not even the number of winning sets needed for a win is stated. The only restriction is that 7 days must be elapsed between two matches against the same players. One of the biggest advantage of using this data set is its similarity to the result database of many professional sports in a given period, due to the large variety of the number of matches between two players and the elapsed time between the matches. However most of the (professional) sports have strong conditions for the opponent selection and the number of matches. It can be assumed that without knowing the organizational rules, the 'opponent selection' in a given period can be regarded as a random process (we note that this is not hold *e.g.* for a Swiss-system chess tournament). Furthermore, we think that the importance of a certain result is inversely proportional to that how old that game is. Thus considering this time-dependency (*i.e.*, the latest results are more important than the older ones) helps to get clearer picture about the actual relative strengths of the opponents.

Results presented here were compared to other traditional and widely used ranking methods. We highlight the advantages of the usage of our method and show its higher predictive power than the other methods. Furthermore, we also suggest a deeper study of a self-organization mechanism respect to the opponent selection: the players having similar tdPR values more likely to play with each other in the later part of the competition (without knowing the scores and ranks of each other). This observation can explain the appearance of different strength classes and emergence of the elite in several sports.

This paper is organized as follows: in Section 2, we give a brief mathematical description of the methods we used and compared, in Section 3, we apply the methods to the table tennis competition and highlight the usefulness and advantages of our approach. finally, in Section 4, we suggest a new type of a self-organization mechanism and a type of graph regularity for deeper analysis.

2 Overview of the ranking methods

In this section we introduce some widely used ranking methods and describe the proposed approach. Hundreds of ranking methods have been appeared in the long history of ranking in sports. The selection of the methods we use in this paper based on a few criterion: (1) the method is based on linear algebra, (2) the method has been proved to be successful for real applications, and (3) the method has simple formulation with closed form solution.

In this section the number of players is denoted by N while we will refer to the players by $1, \dots, N$.

2.1 Least squares method

The first method we describe is usually referred as the least squares (or weighted least squares) method (Lsm) originated from Smith and Gulliksen [15, 30]. Kenneth Massey in his master thesis found wonderful applications of it, especially for ranking the collage football teams in the United States ([21], Chapter 4). The only statistics used by this method are the number of wins and losses of each player. The ranking of the players comes from the solution of the linear system of equations

$$M\vec{r} = \vec{p}, \quad (1)$$

where $\vec{r} = (r_1, \dots, r_N)$ is the unknown rating vector of the players, $\vec{p} = (p_1, \dots, p_N)$ the vector contains the difference of total number of wins and losses for the players, while M (we call it *Massey matrix*) defined as

$$M_{ij} = \begin{cases} n_i, & \text{if } i = j, \\ -n_{ij}, & \text{if } i \neq j, \end{cases} \quad (2)$$

where n_i is the total number of matches played by player i and n_{ij} is the number of matches played between player i and player j . Since $\text{rank}(M) = N - 1$, the linear system Eq. (1) is underdetermined. The non-singularity can be guaranteed if each element of any row i of M is set to 1 and the corresponding p_i is set to 0. Obviously, the decreasing order of the components of the rating vector \vec{r} gives the ranking of the players.

2.2 Colley matrix method

The Colley matrix method was designed by Wesley N. Colley [8]. The method is a modification of the Least squares method by using an observation called Laplace's rule of succession (see [27], page 148) which claims, that if one observed k successes out of n attempts, then $(k+1)/(n+1)$ is better estimation for the next event to be success than k/n . The rating vector \vec{r} of the players is the solution of the linear system

$$C\vec{r} = \vec{b}, \quad (3)$$

where the i th component of the vector \vec{b} is defined as $b_i = 1 + (w_i - l_i)/2$, where w_i and l_i are the number of wins and losses of player i , respectively, and the *Colley matrix* C is

defined as

$$C_{ij} = \begin{cases} n_i + 2, & \text{if } i = j, \\ -n_{ij}, & \text{if } i \neq j. \end{cases} \quad (4)$$

Thus $C = M + 2I$, where I is the $N \times N$ identity matrix. It can be checked that system Eq. (3) always has a unique solution and just as before, the ranking of the players is obtained by the vector \vec{r} .

2.3 Keener method

James P. Keener developed his ranking method [17], based on the theorem of Frobenius and Perron (see e.g. [22], Chapter 8). Using this method, the ranking of the players comes from the eigenvalue equation

$$K\vec{r} = \lambda\vec{r}, \quad (5)$$

where the *Kenner matrix* K defined as

$$K_{ij} = \begin{cases} \frac{w_{ij}+1}{n_{ij}+2}, & \text{if player } i \text{ played against player } j, \\ 0, & \text{otherwise,} \end{cases} \quad (6)$$

where w_{ij} is the number of wins of player i against player j while λ is the dominant eigenvalue (the eigenvalue of the largest absolute value, also known as the *spectral radius*) of the matrix K . The Frobenius-Perron theorem guarantees the existence and uniqueness of the vector \vec{r} with strictly positive components. We mention, that the method has been originally defined for ranking American football teams and used the concrete points that a team i scored against a team j and also used a smoothing function to avoid the possibilities for manipulation. For the table tennis competition that has been examined in this paper, we do not deal with the points scored in the games played just consider the final result of each game as win or loss.

2.4 PageRank method

The PageRank algorithm - developed by Sergey Brin and Larry Page [7] - was originally designed to rank web pages in order to their importance. The idea behind the algorithm came from the basic properties of Markov chains (see e.g. in [27], Chapter 4) as a special case of the Frobenius-Perron theory. The ranking points of the players are iteratively calculated by the recursion formula

$$PR(i) = \frac{\lambda}{N} + (1 - \lambda) \sum_{j \in N^+(i)} \frac{PR(j)}{w_j}, \quad (7)$$

where $N^+(j)$ is the set of players defeated by player i at least once, w_j is the total number of wins of player j and $\lambda \in [0, 1]$ is a free parameter (usually 0.1 or 0.2; the intuitive meaning of λ is described in Section 2.5).

To see the close relationship between PageRank formula and the theory of Markov chains, we write Eq. (7) to the vector equation form

$$\vec{PR} = \frac{\lambda}{N} [I - (1 - \lambda)AD^{-1}]^{-1} \vec{1}, \quad (8)$$

where \vec{PR} PageRank vector contains the PageRank rates of the players, A is the matrix with elements A_{ij} equals to the number of wins of player i against player j , D the diagonal matrix such that $D = \text{diag}[(D_{ii} = \sum_{\ell=1}^N A_{i\ell})_{i=1}^N])$, I is the $N \times N$ identity matrix and finally $\vec{1}$ is the N -dimensional vector having each component equals to 1. Assuming that $\vec{1}\vec{PR} = 1$, Eq. (8) implies, that

$$\vec{PR} = M\vec{PR}, \quad (9)$$

with $M = \lambda/n\vec{1}\vec{1}^T - (1 - \lambda)AD^{-1}$, which shows that \vec{PR} is the eigenvector of the matrix M due to the eigenvalue 1, which is the largest eigenvalue of M by a consequence of the Frobenius-Perron theorem for row-stochastic matrices.

2.5 Time-dependent PageRank method

Intuitively, the basic PageRank algorithm can be considered as a random walk in the graph $G = (V, E)$, where V denotes the set of players and we draw a directed edge $i \rightarrow j \in E$ each time when player i wins against player j . The walk starts in a random node i of the graph and steps to a randomly chosen node j , with uniform probability, for that $i \rightarrow j$ edge exists. The parameter λ can be viewed as a “damping” factor which guarantee that the random walk restarts in a random, uniformly chosen node of the graph almost surely in every $1/\lambda$ -th step. The PageRank of a node i can be considered as the the long-term fraction of the number of visits in node i during the random walk.

Following this direction, we modified the PageRank algorithm such that the weight (i.e the transition probability) of each edge decreases whenever a new edge appears in the graph. Formally, after the k th match was played in a given period, the weight of the latest edge become 1, the second latest become $1/2$, the i th latest become $1/i$, the oldest one become $1/k$. We normalize the weights such that the obtained matrix become row-stochastic (i.e. each row summing to 1) and we recalculate the ranking every time when a new result is registered in the database by solving the equation

$$\vec{PR} = M'\vec{PR}, \quad (10)$$

where the entries of M' are then the new transition probability values, calculated as we described.

3 Experimental results

We applied the methods described above to the table tennis competition of the Institute of Informatics at the University of Szeged (the dataset we used can be found in the website [1]). In that competition, there is no any rule for the selection of the opponents or the date of the match. The only restriction is that 7 days must be elapsed between two matches of the same players. Without considering the organizational rules and by just considering the list of the results in a given period, it can be observed, that these features are occurred in many sports where the competitions are not round-robin.

In Table 1, we report the scores of the players obtained by the different ranking methods. In the case of the PR and the time tdPR algorithms, we used $\lambda = 0.1, 0.2, 0.3, 0.4$,

respectively. Figure 3 shows, that the tdPR score is very robust against these variations of λ . The tdPR method was proved to be very effective in finding the best players of the competition that could be *a posteriori* justified by knowing the players skills.

We used Kendall's τ rank correlation method [18] to quantify the rank correlation between the different methods. The rank correlation coefficient is defined as $\tau = (n_c - n_d) / \binom{n}{2}$, where n_c (n_d) is the number of such pairs that have the same (opposite) order in both ranking list. However, the tdPR score is positively correlated with the win ratio, differences can be seen by comparing the two methods. The relation between the tdPR and the winning ratio is shown in Figure 1(a).

A relevant outlier on the list is player 14 having win ratio 50%, who precedes player 5, 23, 19 and 21 having better win ratio than himself. He is placed at position 4 and this is consistent with the fact, that he was defeated by just that players (player 10, player 12; see the data set and Figure 4) who ranked higher. Figure 1(b) shows the relation between tdPR and the other ranking methods.

Despite the high correlation between tdPR and the other methods, we observed, that the time-dependent method has a better predictive power. We considered the first half of the total number matches had been played since the start of the competition and calculate the tdPR values regarding that period. Then we checked the results of the upcoming matches and the changes in the ranking. It can be observed, that the players with much higher tdPR score after the first half the total matches played, won a high proportion of their matches against players with smaller tdPR values in the later part of the competition. The difference between the tdPR values of the players can give a reliable prediction for the upcoming matches. Figure 2 shows the tdPR ranks of the players after 45, 90 and 180 played games. We mention, that Figure 2 only contains that players, who had already had at least one played matches after the first 45 played matches of the competition. Obviously, at that time we can not predict the results of those players who join later to the competition.

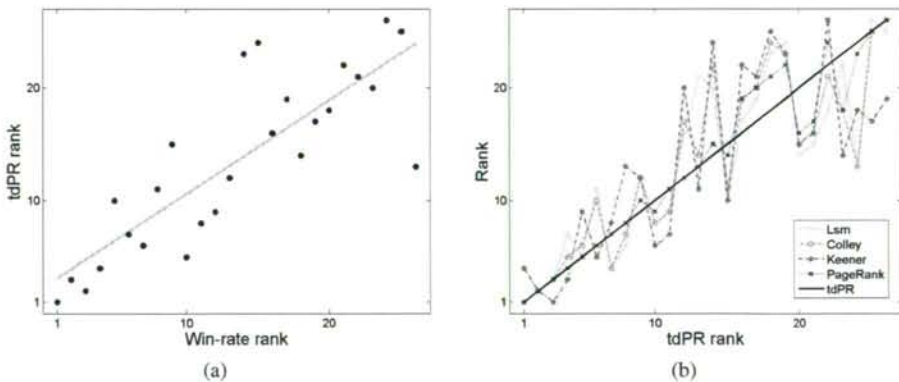


Figure 1: (a) The scatter plot of the tdPR rank vs. the win-rate rank. (b) The results obtained by the different ranking methods.

Table 1: Scores obtained by the different methods; the ordering of the players are obtained by the decreasing order of the tdPR values

Player	#Plays	#Wins	Win ratio	Lsm	Colley	Keener	PR	tdPR
9	13	13	1.000	1.418	1.074	0.229	0.113	0.138
10	29	25	0.862	0.972	0.923	0.238	0.089	0.093
12	30	26	0.867	0.859	0.882	0.245	0.083	0.085
1	63	44	0.698	0.497	0.722	0.233	0.071	0.075
14	6	3	0.500	0.658	0.717	0.198	0.064	0.070
5	38	22	0.579	0.266	0.604	0.200	0.050	0.052
23	5	3	0.600	0.779	0.736	0.199	0.047	0.047
18	16	8	0.500	0.555	0.700	0.192	0.046	0.045
11	24	11	0.458	0.209	0.564	0.193	0.039	0.040
19	10	6	0.600	0.454	0.664	0.200	0.042	0.039
21	13	7	0.538	0.325	0.615	0.199	0.035	0.032
8	19	6	0.316	-0.338	0.354	0.181	0.031	0.032
26	1	0	0.000	-0.503	0.407	0.194	0.031	0.029
4	19	3	0.158	-0.474	0.265	0.172	0.025	0.026
6	10	5	0.500	0.269	0.586	0.194	0.030	0.025
2	17	3	0.176	-0.380	0.307	0.177	0.022	0.024
17	13	2	0.154	-0.437	0.286	0.178	0.019	0.020
3	13	1	0.077	-0.615	0.213	0.171	0.019	0.020
7	12	2	0.167	-0.650	0.219	0.176	0.018	0.018
16	2	0	0.000	-0.322	0.401	0.191	0.024	0.018
13	2	0	0.000	-0.322	0.401	0.191	0.024	0.018
22	14	1	0.071	-0.433	0.277	0.169	0.016	0.016
24	4	1	0.250	-0.507	0.349	0.191	0.023	0.016
15	5	1	0.200	-0.174	0.416	0.188	0.017	0.010
25	3	0	0.000	-1.060	0.186	0.191	0.015	0.007
20	5	0	0.000	-1.047	0.136	0.184	0.010	0.004

Table 2: Kendall's τ rank correlation between the different methods.

	Win/loss	Lsm	Colley	Keener	PR	tdPR
Win/loss	1.000					
MASSEY	0.705	1.000				
COLLEY	0.748	0.895	1.000			
KEENER	0.655	0.606	0.711	1.000		
PR	0.723	0.735	0.803	0.662	1.000	
tdPR	0.723	0.674	0.705	0.563	0.902	1.000

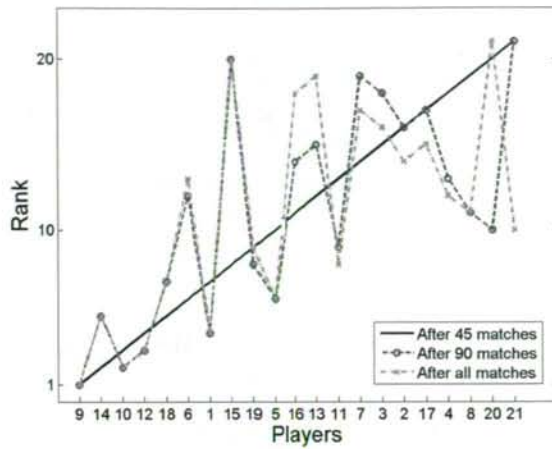


Figure 2: The tdPR ranks of the players after 45, 90 and 180 played games.

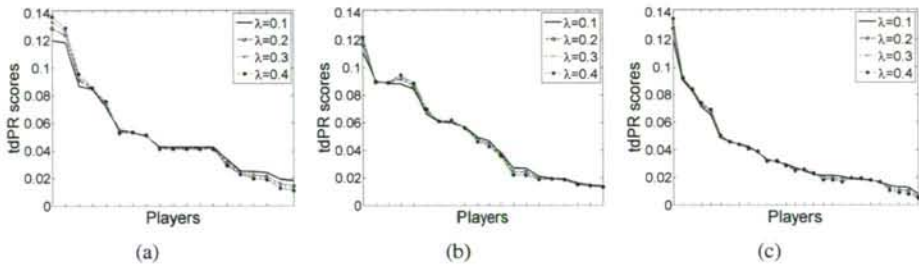


Figure 3: Sensitivity analysis of tdPR for different λ values after 45, 90 and 180 played games, from left to right. The figure shows, that the tdPR score is robust against these variations of λ .

4 Further ideas and future work

We also ran a clustering algorithm (aiming to maximize modularity [25]) to see whether there exists a deeper organizational mechanism behind the evolution of such a network. In Figure 2 the clusters are colored with different colors. Figure 4 illustrates the contact graph of the players after 90 played matches (left hand side) and the current state of the championship with more than 180 matches (right hand side). It is interesting to see the changes of the clusters of the two graph. First, we observed that most of the new players wants to play against the actual best players (in tdPR rank) hoping to jump to the top of the ranking table. Second, it seems that players having closer tdPR values more likely to play with each other, than players having much less tdPR value and rank. Thus, we conjecture that the tdPR scores have a good explanatory power for a self-organizing mechanism of free-time sports and it can explain the appearance of different strength classes in most

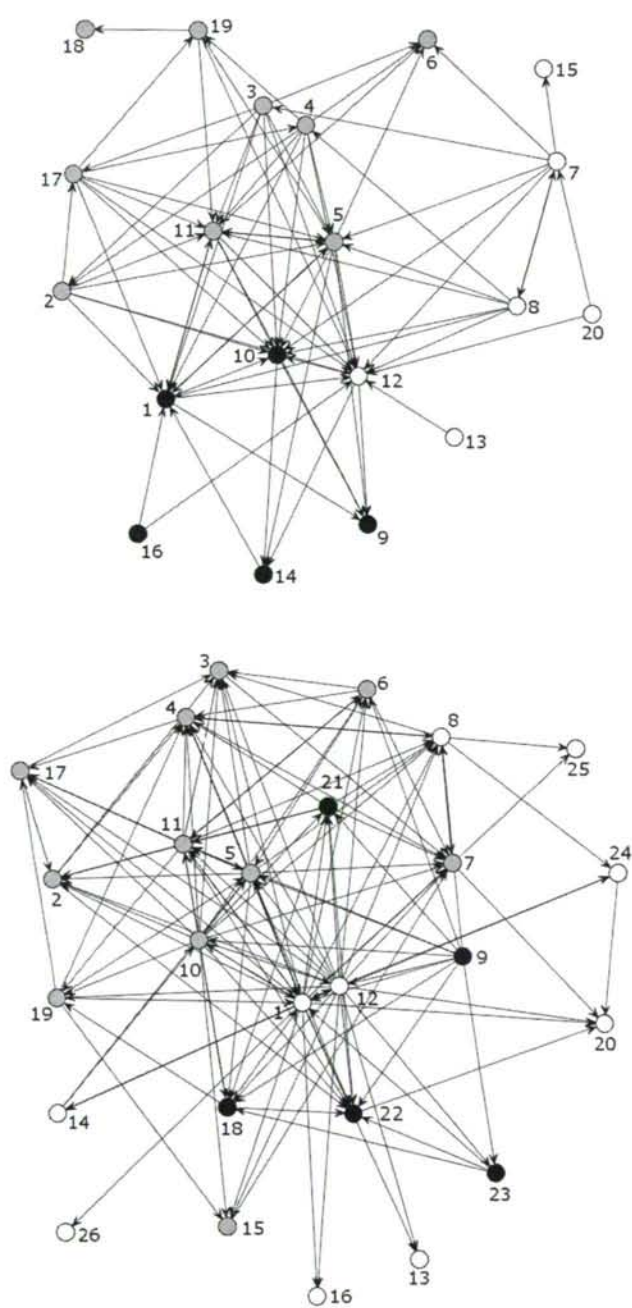


Figure 4: The contact graph of the players after 90 played matches (top) and the current state of the championship with more than 180 mathces (bottom). Nodes having same colors belong to the same clusters.

of the sports, where the results in a class are more difficult to be predicted than results between different classes. Furthermore, in a graph theoretical point of view, a new type of 'regulatory' (for some details, see [9]) can be defined on directed graphs, where the fraction of in/out edges of a node is around $1/2$ in the same class, and tends to 1 (or 0 reversely) between different classes.

5 Conclusions

Graph based algorithms have been proved to be relevant in a wide range of applications. However there is no perfect algorithm for ranking sport players/teams, we believe that PageRank based methods are reliable to ranking athletes and this is even more true for time-dependent modifications of these stochastic algorithms.

In this work, we defined a time-dependent PageRank based algorithm and applied it for ranking players in a university table tennis competition. According to our tdPR method, the ranking of a player is not only determined by the number of his or her victories, but matters from how good players he could beat or lose against. It means, that a good player is needed to beat for higher ranking position, but win many matches against weaker opponents does not lead anyone to the first positions in the ranking table. The time-dependency of weights of the matches guarantee that the matches played a long time ago do not count as much weight in the ranking. Another aim of the time-dependency is to pressure the players to play regularly or else their results would be out of date, therefore count much less in the ranking.

We also observed that our method has a good predictive power. This can be interesting in other aspects of sports, for example estimate the betting odds for games. Finally, we think that a self-organization mechanism works in the background of the evolution of the contact graph. Obviously, players want to enter matches are expected to be exciting, but this nature of such competitions can be modeled and measured mathematically just by knowing the time-series of the results. That observation gives the idea to define a special preferential attachment mechanism [24] where players having higher PageRank values more likely to play (contact) with each other and this is maybe related to the emergence of an elite in sports. Further research is needed around this hypothesis, and testing our method for different sports and data sets is also another work for the future.

Acknowledgment

The first author was supported by the **European Union** and the **State of Hungary, co-financed by the European Social Fund** in the framework of TÁMOP-4.2.4.A/2-11-1-2012-0001 'National Excellence Program'.

The authors also thank Tibor Csendes and András Pluhár for useful discussion and every excellent researcher of the Informatics Institute of University of Szeged who take time to relax at the ping-pong table. We are indebted to the referees for their careful reading and numerous suggestions which improved the presentation of the paper.

References

- [1] <http://www.inf.u-szeged.hu/~london/TableTennisResults.txt>.
- [2] Baeza-Yates, Ricardo, Saint-Jean, Felipe, and Castillo, Carlos. Web structure, dynamics and page quality. *String Processing and Information Retrieval, Lecture Notes in Computer Science*, 2476:117–130, 2002.
- [3] Bahmani, Bahman, Chowdhury, Abdur, and Goel, Ashish. Fast incremental and personalized pagerank. *Proc. VLDB Endow.*, 4(3):173–184, December 2010.
- [4] Berkhin, Pavel. A survey on PageRank computing. *Internet Mathematics*, 2(1):73–120, 2005.
- [5] Boginski, Vladimir, Butenko, Sergiy, and Pardalos, Panos M. Matrix-based methods for college football rankings. *Economics, Management and Optimization in Sports*, pages 1–13, 2004.
- [6] Bradley, Ralph Allan and Terry, Milton E. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- [7] Brin, Sergey and Page, Lawrence. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.
- [8] Colley, Wesley N. Colley’s bias free college football ranking method: the Colley matrix explained. <http://www.colleyrankings.com/matrate.pdf>, 2002.
- [9] Csaba, Béla and Pluhár, András. Weighted regularity lemma with applications. *arXiv preprint arXiv:0907.0245*, 2009.
- [10] Csató, László. Ranking by pairwise comparisons for Swiss-system tournaments. *Central European Journal of Operations Research*, 21(4):783–803, 2013.
- [11] Dahl, Geir. A matrix-based ranking method with application to tennis. *Linear Algebra and its Applications*, 437(1):26–36, 2012.
- [12] David, Herbert A. Ranking from unbalanced paired-comparison data. *Biometrika*, 74(2):432–436, 1987.
- [13] Govan, Anjela Yuryevna. Ranking theory with application to popular sports. *PhD dissertation, North Carolina State University, Raleigh, North Carolina*, 2008.
- [14] Govan, Anjela Yuryevna, Langville, Amy N, and Meyer, Carl D. Offense-defense approach to ranking team sports. *Journal of Quantitative Analysis in Sports*, 5(1):1–19, 2009.
- [15] Gulliksen, Harold. A least-squares solution for paired comparisons with incomplete data. *Psychometrika*, 21(2):125–134, 1956.

- [16] Jech, Thomas. The ranking of incomplete tournaments: A mathematician's guide to popular sports. *The American Mathematical Monthly*, 90(4):pp. 246–264+265–266, 1983.
- [17] Keener, James P. The Perron-Frobenius theorem and the ranking of football teams. *SIAM Review*, 35(1):80–93, 1993.
- [18] Kendall, Maurice G. A new measure of rank correlation. *Biometrika*, 30:81–93, 1938.
- [19] Kendall, Maurice G and Babington Smith, B. On the method of paired comparisons. *Biometrika*, 31(3/4):324–345, 1940.
- [20] Langville, Amy N and Meyer, Carl D. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–380, 2004.
- [21] Massey, Kenneth. Statistical models applied to the rating of sports teams. *Master thesis, Bluefield College*, 1997.
- [22] Meyer, Carl D. *Matrix analysis and applied linear algebra book and solutions manual*, volume 2. Society for Industrial and Applied Mathematics, 2000.
- [23] Motegi, Shun and Masuda, Naoki. A network-based dynamical ranking system for competitive sports. *Scientific Reports*, 2:904, 2012.
- [24] Newman, Mark EJ. Clustering and preferential attachment in growing networks. *Physical Review E*, 64(2):025102, 2001.
- [25] Newman, Mark EJ. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the USA*, 103(23):8577–8582, 2006.
- [26] Radicchi, Filippo. Who is the best player ever? A complex network analysis of the history of professional tennis. *PloS ONE*, 6(2):e17249, 2011.
- [27] Ross, Sheldon M. *Introduction to probability models*. Academic Press, Ninth edition, 2007.
- [28] Rossi, Ryan A and Gleich, David F. Dynamic pagerank using evolving teleportation. In *Algorithms and Models for the Web Graph, Lecture Notes in Computer Science*, volume 7323, pages 126–137. Springer, 2012.
- [29] Sargolzaei, P and Soleymani, F. Pagerank problem, survey and future research directions. *International Mathematical Forum*, 5(19):937–956, 2010.
- [30] Smith, John H. Adjusting baseball standings for strength of teams played. *American Statistician*, 10(3):23–24, 1956.

Received 13th November 2013

CONTENTS

Programming Languages and Software Tools	287
Preface	289
<i>Jari-Pekka Voutilainen, Anna-Liisa Mattila, and Tommi Mikkonen:</i> Lively3D: Building a 3D Desktop Environment as a Single Page Application	291
<i>Antti Valmari:</i> Asymptotic Proportion of Hard Instances of the Halting Problem	307
<i>István Kádár, Péter Hegedűs, and Rudolf Ferenc:</i> Runtime Exception Detection in Java Programs Using Symbolic Execution	331
<i>Gergő Gombos, Tamás Matuszka, Balázs Pinczel, Gábor Rácz, and Attila Kiss:</i> VOSD: A General-Purpose Virtual Observatory over Semantic Databases	353
<i>Vard Antinyan, Mirosław Staron, Jörgen Hansson, Wilhelm Meding, Per Österström, and Anders Henriksson:</i> Monitoring Evolution of Code Complexity and Magnitude of Changes	367
<i>Otto Hylli, Samuel Lahtinen, Anna Ruokonen, and Kari Systä:</i> Service Composition for End-Users	383
<i>Ákos Hajdu, András Vörös, Tamás Bartha, and Zoltán Mártonka:</i> Extensions to the CEGAR Approach on Petri Nets	401
<i>Richárd Dévai, Judit Jász, Csaba Nagy, and Rudolf Ferenc:</i> Designing and Implementing Control Flow Graph for Magic 4th Generation Language	419
<i>Ferenc Horváth, Szabolcs Bognár, Tamás Gergely, Róbert Rácz, Árpád Beszédes, and Vladimir Marinkovic:</i> Code Coverage Measurement Framework for Android Devices	439
Regular Papers	459
<i>Gábor Recski:</i> Hungarian Noun Phrase Extraction Using Rule-based and Hybrid Methods	461
<i>Miklós Ujvári:</i> Applications of the Inverse Theta Number in Stable Set Problems	481
<i>András London, József Németh, and Tamás Németh:</i> Time-dependent Network Algorithm for Ranking in Sports	495

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János